BUILT-IN SELF-TEST FOR AN AIRBORNE DIGITAL COMPUTER

Edward Oliver Bierman

United States
Naval Postgraduate School

THESIS

Built-In Self-Test for an Airborne Digital Computer

by

Edward Oliver Bierman

June 1969

Built-In Self-Test for an Airborne Digital Computer

by

Edward Oliver Bierman
Major, United States Marine Corps
B.S., United States Military Academy, 1960

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 1969

ABSTRACT

This thesis describes the design of a built-in self-test capability for a military airborne digital computer. The supportive investigation of program constraints and their effects on the example test design is intended to give broad perspective to the general self-test design problem. Alternate procedures for achieving the goal of airborne detection and isolation of a certain class of failures to the modular level are surveyed. A specific test design is evolved illustrating the unique mix of program-oriented, periodic techniques, and added hardware, continuous techniques best suited to the example development program. The test design is evaluated and further work is suggested.

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENT

The author wishes to acknowledge his considerable debt to the Advanced Projects Laboratory, Data Systems Division, Aerospace Group, Hughes Aircraft Company, Culver City, California, in the preparation of this thesis. In particular, Mr. Charles A. Pullen of the Advanced Computer Systems Department, whose competent guidance and constructive suggestions throughout the major portion of the supporting project were so helpful, should be cited. In addition, the assistance of Mr. Charles B. Ames, Mr. Noel A. Boehmer, Mr. Donald Achterberg, Mr. Charles Disparte, Mr. John A. Schiro, and Mr. Lynn Anderson, all of Hughes Aircraft Company, is also gratefully acknowledged.

The role played by Professor Mitchell L. Cotton of the Naval Postgraduate School in the writing of this thesis should not go unmentioned. His assistance and advice in defining the investigation, watching its direction, and maintaining its perspective throughout is sincerely appreciated.

I. INTRODUCTION

Maintenance and repair of faulty electronic equipment have always been the less glamorous companions of design and operation. Indeed, the subjects were often broached only after design concepts were formed and specific circuitry developed. The evolution of increasingly complex electronic systems, such as digital computers, has forced greater and earlier consideration of the problems of locating failures and correcting them. A digital computer which automatically tests itself for proper operation and which provides valuable information to facilitate maintenance and repair has become very attractive for military and space systems applications. This thesis reports the results of an investigation to provide such automatic self-checking for a digital computer system.

A project which considerably supported the investigation was accomplished at Hughes Aircraft Company, Culver City, California, during an industrial experience tour. The project goal of designing a built-in self-test (BIT) capability for an advanced airborne digital computer system for military application was more fully realized because BIT was accepted as a principal design consideration early in the architectural design procedure. The specific design developed will be used as an example; however, the test procedures will be recognized as being more generally applicable to the class of digital computers for which the assumptions and constraints applied herein can be validated. Only one of many possible solutions to the fault detection and isolation problem will be presented. The choice made should not be construed to reflect official policy at Hughes Aircraft Company.

Some general comments at the outset should place this investigation in proper perspective and temper expectation with pragmatism. The investigation has as its central focus the specific BIT design developed; however, it is intended to consider the broader systems design options available, thereby showing the example design in better perspective. As Sellers, Hsiao and Bearnson [Ref. 43] so aptly observe, one should initially set reasonable design objectives relative to the thoroughness of test, recognizing that exhaustive automatic test is an almost unattainable practical goal. As part of a computer development program, the BIT design is subject to the larger program objectives and constraints. The first part of this investigation will define the test design problem in more specific terms. Subject to practical limitations, a reasonable set of test objectives will be developed. Once objectives have been focused, alternatives for implementation will be considered and a test concept evolved. Specific test procedures will be presented for automatically testing the digital computer. Finally, the results obtained will be critically evaluated in light of the design objectives, and further related work will be suggested.

II. PROBLEM DEFINITION AND DESIGN OBJECTIVES

A. BROAD GOALS

Given the framework of a digital computer in a military avionics application, one can identify three broad goals for a self-test capability:

1. To decrease the cost of ownership by reducing maintenance cost/-time and increasing system availability.
2. To indicate to the pilot in flight the level of system operational capability available to him.
3. To provide limited assistance through self-test in prototype design and checkout.

Any information relative to the existence and location of failure will reduce the time spent (and hence cost) to repair the computer and therefore increase the aircraft's availability for operational purposes. Airborne indications of system degradation through failure allow the pilot to make timely and informed choices of alternatives to optimize the probability of successful mission completion. Lastly, self-test during computer development assists the engineer to more quickly identify and correct design and hardware faults. In short, BIT is designed to provide a greater system effectiveness at a lower cost; that is, to increase cost-effectiveness.

B. PROGRAM CONSTRAINTS

1. Cost of BIT

In a very real sense, the dominating factor effecting BIT design problem definition is cost. Cost has several facets. The cost of BIT is considered to be part of the overall computer program price tag. Required performance criteria for the completed computer system are

specified by the sponsoring government agency to the aerospace industry. A participating company must strive to reduce its proposed system's cost while meeting or exceeding specifications to remain competitive. So within the overall program development and production cost, the contribution of BIT must be justified and minimized. Since the broad goal of increased cost-effectiveness has been identified for BIT, justification includes critical assessment of the added cost to the computer program of providing a self-test capability to ensure that a compensatory benefit in reduced cost of ownership will be realized.

Sources of added cost for BIT include but are not limited to the following:

1. The checking hardware itself
2. Additional power required
3. Greater capacity logic to provide for the added checking hardware; e.g., drivers with greater fanout
4. Additional data lines to provide for test hardware and procedures
5. Storage capacity required for BIT routines and data
6. Design, programming, and development costs

Other "costs", often translated into dollar values, include the penalties (if any) attached to increased size and weight of an airborne computer provided with BIT capability. For an air superiority fighter application, these penalties are severe.[1]

---

[1] Hughes Aircraft Co. uses internally generated weighting factors of \$500/lb and \$5000/ft^3 for added hardware. To illustrate using these typical penalties, two computers are compared:

1) A 0.5 ft^3, 25 lb computer costing \$50k
2) A 0.4 ft^3, 20 lb computer costing \$52.5k

The penalties added to computer (1) are:

0.1 ft^3 x \$5000/ft^3	=	\$ 500	for volume
5 lb x \$500/lb	=	\$2500	for weight
Total	=	\$3000	penalty

Computer (2), though ostensibly costing more, is \$500 less expensive after penalties are applied.

The benefits of BIT can also be reduced to monetary terms by operations analysis techniques. Projected maintenance experience, spare parts costs, inventory levels, and the effects of maintenance concepts can all be given dollar values. However, the relative weight that increased system operational availability receives is more subjective. In a space system, for example, there is a very high premium on availability; in a military airborne system, availability is important but not as critical.

The result on the overall cost of ownership for the military system is that, while the penalties for providing BIT are quite clear, the benefits are harder to evaluate and therefore less visible. Even when a clear long-term reduction in cost of ownership can be expected, insufficient available funding may force procurement of a less expensive option without a BIT capability. The effect on BIT design is to place emphasis on minimizing the more visible penalties, reducing them to an acceptable fixed percentage of the system cost without a BIT capability.[2]

2. The Parent Computer

The nature of the computer for which the self-test capability is to be provided certainly has a large influence on the BIT design objectives. For the example design, the characteristics for the parent computer evolved from the original specifications and the subsequent company policy decisions. The parent computer was to:

---

[2]Estimates in the literature range from 3% cost increase for BIT for a commercial machine to over 300% for a triplicated space system computer. A figure of 10% fell in the general area of acceptability at Hughes Aircraft Company for this project.

1. Have a military avionics application
2. Be modular
3. Have flexible word length
4. Be non-redundant
5. Be repaired on the ground, not in the air
6. Have minimal storage capacity
7. Suffer no operational degradation because of BIT
8. Be developed on short schedule at low risk

Each of these characteristics will be more thoroughly discussed.

A military avionics application implies that size and weight are to be minimized consistent with the cost penalties discussed earlier. It also implies high speed, real-time computation. The more rigid military specifications concerning operating temperatures, humidity, shock resistance and other severe environmental factors affect the quality of components used and the packaging of these components at all levels.

The computer was to be of modular construction, the term module referring to a standardized plug-in circuit card with a given surface area and number of pin connectors. The Naval Avionics Facility Indianapolis (NAFI) has developed a series of modules designed to be acceptable as the basic building blocks for many military applications [Ref. 10]. The basic "NAFI module" chosen for use in the parent computer (with some modifications) was the "2A" size whose important features relative to BIT design are dimensions of roughly five (5) inches in length and two (2) inches in height (both sides may be used for mounting hardware) and 80 pins in the two bottom connectors. Figure 1, derived from Ref. 10, depicts the 2A NAFI module. The module's surface area and number of pins place limitations on (1) the amount of hardware

which will physically fit on the module (heat dissipation is a related problem), and (2) on the number of external, intermodular electrical paths available. The level of solid state technology of the implementing circuitry determines whether the area or pin limitation dominates. For example, circuitry consisting of discrete components (separate transistors, capacitors, resistors) tends to impose an area limitation because the relatively large size of individual components limits the number which can be accommodated in the fixed area, before the available pin connectors are exhausted. At the other extreme, circuitry implemented using large scale integration (LSI) technology, in which perhaps 1000 or more gates are placed on a single silicon chip [Ref. 48], requires little mounting surface area. The number of external connections needed, however, can be large. Hence, in the latter case a pin limitation exists. In between these extremes fall the integrated circuit (IC) and medium scale integration (MSI) technological levels which may be area or pin limited for specific modules. The size of the modular partition chosen for the parent computer and the predominantly IC/MSI technology utilized will be seen to have a significant effect on BIT design.

Partitioning of the parent computer was not otherwise specified, except that the computer's basic design was to be readily adaptable for differing word length applications (specifically, multiples of eight bits, up to a 32-bit word length) without major redesign of the original modules. The expected initial application of the parent computer specified a 24-bit word length; this word length will be used in the example design.

The parent computer was to be essentially non-redundant; that is, no general replication of hardware at any level was intended. This constraint arose from cost considerations. Penalties in the additional hardware cost, increased size and weight associated with redundancy were deemed unacceptable. Additionally, the mean time between failures (MTBF) of the computer tends to be several times higher than the MTBF of the equipment which the computer serves; e.g., a radar.[3]

A closely related characteristic dictated ground repair of failures. No automatic reconfiguration under failure or fault-masking was intended, since such self-repair generally requires some redundancy. Airborne personnel to effect maintenance would not be available in the type aircraft for which application was projected. Access, removal of shielding, and dust-free repair would be difficult airborne. Built-in test was therefore restricted to detection and isolation of faults, and was not intended to include a self-repair capability.

The requirement for minimal storage capacity was again related to cost. Random access storage such as core memory is expensive in hardware, size, weight, and power requirements. No peripheral bulk storage devices such as drum, disc, or tape were to be available. The effect of these characteristics of the parent computer on the design of BIT is significant. The dedication of memory bit locations to storage of error detecting codes, such as parity or residue, is eliminated from consideration because of the attendant reduction in word

---

[3]Reference 34 shows MTBF's in the 100's of hours for the F-111A weapons system avionics equipments. MTBF's for airborne computers, as shown by marketing brochures, are typically in the 1000's of hours.

length available to the flight program. Increased word length is unacceptable because of the greater storage requirement and higher cost. Coding is a widely used technique for detecting data transfer errors. The storage of software self-test programs and data in core-memory is also virtually eliminated from the list of often-used test tools. The core memory, then, is reserved for the flight program and for operational use with negligible capacity available for BIT use.

Any self-test capability is not allowed to degrade the real-time operational efficiency of the computer in speed or availability. The effect of this requirement is to prohibit the insertion of test hardware in operational propagation paths because of the delays thereby introduced. Additionally, any sequential, program-oriented test routines would have to be exercised on a time-shared basis with ongoing tactical operations in available short blocks of "idle" time. Such routines would therefore have to be interruptable without destroying test efficacy so that the machine could be returned to operational computation immediately, whenever required.

The overall computer program called for a short development schedule with low risk to the company. These constraints dictate the use of existing techniques and designs wherever feasible. No completely new technology could be developed within schedule requirements. Off-the-shelf hardware components would be primarily used because of the risks attendant in meeting a short schedule with components potentially available from outside suppliers at production time but still under development during computer design.

C. BIT DESIGN OBJECTIVES

With the aforementioned broad goals set forth and the constraints imposed on self-test design by the nature of the larger program more clearly defined, realistic BIT design objectives can be developed. The maintenance problem would be most significantly assisted if faults could be isolated to the plug-in card, or modular level. Sub-modular fault isolation, while desirable from the standpoint of higher echelon maintenance, does not contribute any more significantly to increased aircraft availability since the faulty module must be removed in either case. Conformal coating for environmental protection applied to circuitry within the module makes removal of sub-modular components a difficult and specialized task inappropriate at the immediate squadron (1st echelon) level. Higher level isolation would require replacement of large and more expensive units of the computer. Stocking of spare parts at the module level seems reasonable for the squadron shop both in the inventory costs involved and the volumes required. Of course, commonality among modules reduces the different types to be stocked and is desirable. These heuristic arguments can be quantized, but the views presented should suffice to intuitively support the decision to set fault isolation to the modular level as a BIT design objective.

Since no airborne repair, manual or automatic, is required, reporting of faults detected within specific modules completes the self-test task. A compatible design objective, supporting the second and third broad goals related to pilot notification of failure and aid to prototype development, is to rapidly indicate the specific modular

location of existing faults to a central location for immediate use by the pilot and later use by maintenance personnel upon mission termination.

The BIT design objectives and major constraints can now be summarized. The BIT design should automatically detect failures in the computer and isolate them to the modular level airborne. The modular location of such failures should be rapidly reported to a central location. The design should be minimized as to cost, require negligible core memory storage, utilize no coding techniques requiring storage capacity, and inflict no operational degradation on the computer's speed and availability. All this should be accomplished on short schedule and at low risk. While these objectives and constraints for a self-test design are imposing, they are not atypical of the requirements of a military airborne system. Just what constitutes the failure to be detected can now be examined.

III. THE NATURE OF FAILURE

Since the objective of "fault detection" has been set, its meaning should be explained. This section will consider what constitutes a fault and will define several related terms. The literature is replete with descriptive terms such as catastrophic, intermittent, solid, transient, burst, marginal, multiple, insipient, minor, and gross, applied to fault and the related terms failure, error and malfunction.[4] The terms "fault," "failure," and "malfunction" will be used synonymously to mean a physical defect in equipment which causes that equipment to perform in an unsatisfactory manner. The substandard performance usually resulting from a fault will be termed an "error." Another way of stating this is to say that an error is an incorrect result. The terms "solid" and "intermittent" will be used to characterize the duration of the error, and by inference, the failure causing the error. A solid error will refer to an error which results from a failure which persists; a solid error will consistently recur under the same equipment conditions. An intermittent error will be one which is of short or transient duration and is non-persistent; that is, an intermittent error does not consistently recur given the same conditions. The terms "catastrophic" and "transient" are often used to describe these two categories of error, but they will not generally be used herein. The idea of degrees of failure is introduced by such terms as marginal,

---

[4] A good discussion of some typical terminology surrounding "failure" is found in Ref. 24.

single or multiple, minor or gross. The term "marginal" will be reserved to describe a category of testing. The terms "single" and "multiple" will refer to one failure or error, and to more than one failure or error, respectively.

Erroneous results can arise from sources other than equipment failure. Programming inaccuracies and human operator mistakes will not be considered to be error within the scope of this investigation. Equipment failure leading to erroneous results represents the class of faults to be detected by the design test techniques. Inaccurate intra-computer data transmission, faults in logic, failures in core memory, and failed test circuitry are representative of faults within this class of interest.

Certain types of equipment, generally termed "hard-core," serve the entire computer and must operate properly if the computer is to function at all. Examples of such equipment are main power supplies, clocking circuitry, cooling equipment and other mechanical components such as electromagnetic interference shielding. Faults in this hard-core equipment have been effectively identified by voltage/temperature sensing devices which continuously compare performance to preset tolerances, and similar well-known techniques [Ref. 46]. Faults in the types of hard-core equipment described above will not be considered to be part of the BIT detection and isolation task as defined herein. The main thrust of this investigation will treat the less adequately resolved problems of identifying and locating all possible failures in the logic

circuitry, storage, data transmission paths, checking hardware and other equipment which is not hard-core in the previous sense of providing "housekeeping" and utility services.[5]

Faults are usually identified by detecting the resultant errors. If a fault does not produce erroneous results, its existence is of little immediate consequence. For example, a shorted transistor always causing an output to be in the low voltage level (the zero of positive logic having the binary logical states one and zero) does not become significant until the high voltage level represents the proper output value. In other words, a stuck-at-zero failure is not important until the proper result should be a logical one. Conversely, as previously mentioned, all errors are not the result of equipment failure (e.g., operator mistakes), but some of these appear to be the result of equipment failure. Equipment failure modes should be examined to identify those of interest to the test design.

Assuming transistor building blocks (discrete, IC, MSI, or LSI technology) for the example computer logic (vice cryogenics or some other technology), some of the possible failure modes are:

1. Inputs or outputs stuck at the high or low voltage levels (stuck-at-one, stuck-at-zero). Inputs stuck above the high level or below the low level, a possible condition in some computers, have the same effect.
2. Inputs or outputs stuck at an indeterminate, intermediate level between the high and low voltage levels. Indeterminate voltage levels might sometimes be interpreted as a one, and sometimes as a zero.

---

[5]The term "hard-core" will later also be applied to some equipment within this group subject to test, but in a different sense.

3. Deteriorated component response to inputs or weakened drive capability of outputs.

The first failure mode is the one of greatest interest for the subject test design because such persistent failures result in solid errors susceptible to detection and isolation.

The second failure mode, inputs or outputs stuck at an indeterminate voltage level, might lead to no error if properly interpreted, intermittent error if interpreted differently at different times, or solid error if consistently misinterpreted. An assumption which is often made in deriving a diagnostic scheme is to disallow the second failure mode.[6] Another way of stating this is to assume that logic fails to one of the two logic levels, one or zero, and not to some intermediate level. The assumption can be validated by setting a voltage threshold above which results will be interpreted as one logical state, and below which results will be interpreted as the other logical state. The assumption of disallowing the second failure mode will be made for the test design.[7]

The third failure mode could result in solid or intermittent errors depending on the consistency of the erroneous results and the duration. For example, a weak driving capacity of an output feeding several subsequent inputs (fan out) could result in some inputs receiving a zero and others a one. This would be a solid error if the same

---

[6] For example, see Ref. 31.

[7] This assumption is occasionally not made. For example, one scheme which relies on circuitry which fails to a NULL state intermediate between one and zero is described by Connolly and Schmitt [Ref. 8]. The assumption of failure to one or zero is far more common.

inputs always received the same signal under the given conditions. An intermittent error would result if, for example, a driven input received a logical one in one instance and a zero in another for the same driving output value. The third failure mode is considered part of the test problem. It will be discussed again under the topic of marginal testing in Section IV.

Intermittent errors should be discussed more fully, as they are sometimes part of the test problem and sometimes not. Some physical causes of intermittent errors are:

1. Dirty connectors - a small smudge of oil or dirt on a pin might be sufficient to intermittently block the low current levels typically found in intermodular lines. Vibration can provide slight shifts in the contact surfaces sufficient to make or break contact.
2. Temporary overheating of hardware regions - when not persistent, such transient environmental conditions can cause intermittent erroneous results.
3. Loose connections or particles between circuits or within hardware packages - vibration can cause open and closed circuit conditions intermittently.
4. Unusual electromagnetic interference (EMI) or coupling-spikes coupled into the circuitry from outside, or appearing through the power supply can cause changes in state resulting in erroneous performance.
5. Drifting characteristics - aging or deteriorating components or changing environmental conditions can cause varying and inconsistent performance changes in circuitry.

While the above list is certainly not complete, it does serve to illustrate the many sources of intermittent error, and to suggest the difficulty of detecting and isolating the causes of such errors. Those causes not representing hardware failure, such as dirty connectors or unusual EMI, can cause erroneous results which falsely indite fault-free circuitry (which, when faulty, exhibits the same symptoms). Such causes of faulty performance are important because even one state

change affecting a logical decision within the machine can produce catastrophic results. While intermittent errors caused by other than hardware failure have been excluded from the test problem, test procedures must endeavor to ensure they are, in fact, excluded. A procedure which signals hardware failure when none exists not only reduces the level of confidence accorded error signals, but also increases cost, in direct opposition to BIT objectives, by causing fault-free circuitry to be replaced.

The degree, or extent, of failure is also important to test design. Single failures are inherently easier to detect and isolate than multiple failures; the detection problem is smaller. Additionally, multiple solid failures can have the property of occasionally masking each other, giving the appearance of intermittent single failure. To reduce the test problem to reasonable limits, the assumption that there exists at most a single failure in a computer to be tested is often made. The validity of the "single failure assumption" will be examined relative to the example BIT design as a possible means of reducing the quantity of added hardware required to give sufficient test effectiveness within acceptable program bounds.

The components used in modern military/space systems are designed to have high individual component reliability. Low power silicon transistors in the Raytheon equipment used in Apollo and Polaris programs, for example, were found to have a failure rate of 1.4×10^{-5} failures/1000 hours [Ref. 40]. If multicomponent packages such as IC's are used, the interconnections between components on the same silicon chip are more reliable than in the discrete component case. Overall

equipment reliability can therefore be expected to go up through the use of integrated circuits [Ref. 29]. Figures provided from a variety of aerospace suppliers 1964 to 1966 show failure rates for integrated circuits from 7×10^{-5} failures/1000 hours to 5.2×10^{-4} failures/1000 hours [Ref. 19]. Brauer has reported integrated circuit failure rates varying from 7×10^{-6} failures/1000 hours to 6×10^{-3} failures/1000 hours [Ref. 4]. Infant mortality failures and adolescent failures, usually occurring during burn-in and testing at the factory, exceed the exponential failures (constant failure rate) more common in an operationally deployed unit. This partially accounts for the diversity in the cited failure rates, and emphasizes the need to know failure rate sources and conditions for proper interpretation. The point to be made is that even the most pessimistic of the cited figures shows that a long operating life can be expected from modern components.

The MTBF of a computer considers all the different component failure rates in addition to connection reliabilities and workmanship flaws in assigning a commonly used overall reliability figure of merit. The MTBF of the digital airborne computer can be expected to be in the 1000's of hours.[8] With system MTBF's of this order of magnitude, the probability of experiencing one failure in a short time interval is very small. Experiencing two or more failures in the same short time interval is highly improbable. It then seems reasonable that one incurs a very small risk of undetected error if one designs test techniques

---

[8]The Autonetics D26J airborne computer with an estimated MTBF of 18,000 hrs; the Litton LC-728, 4,250 hrs; the Raytheon R-11, 3,500 hrs; the CDC 5400, 2,500 hrs are examples from marketing brochures.

assuming single failure, as long as testing is done at least periodically at short intervals. This intuitive approach is used, as more exact calculations are dependent on actual failure rates, numbers and types of components, specified confidence levels and assumed distributions. The single failure assumption seems to be justified for the example design, and will be made. Restated, the assumption asserts that the computer is constructed of highly reliable individual components so that essentially simultaneous failure of more than one component is so improbable that it can be reasonably neglected. The assumption is further justified economically by program limitations in that testing for multiple failures requires more added hardware at an unacceptable cost penalty.

The foregoing examination of the nature of failure has led to some assumptions and conclusions relative to BIT design. First of all, logic will be assumed to fail to one of its two logic states, and not to some intermediate level. Solid failures will be of major interest; however, any failure leading to erroneous results is part of the detection and isolation problem. Intermittent errors will be especially difficult to detect and isolate. Those erroneous results caused by non-hardware sources are important in that care must be taken to avoid condemning fault-free hardware as their source. Finally, the single error assumption will be made because little risk of undetected error is thereby incurred, and it presents the most reasonable approach from an economic standpoint. Now the possible test procedures available to meet the BIT objectives can be considered.

IV. TEST PROCEDURE ALTERNATIVES

A. GENERAL CONSIDERATIONS

Comparison forms the basis of all test procedures. A norm against which comparison can be made must be available, either a priori or as a result of some generating process. The computer then produces a result which is suspect until verified against the norm. The variety of procedures available for testing a computer have this comparative process in common.

Since thorough testing for all possible errors within the test area of interest is the objective, the different levels at which testing can be conducted should be identified. The computer can be functionally exercised by directing it to perform the operations for which it was designed on a variety of operands. The thoroughness of test can be evaluated by asking how many of the possible machine states are thereby verified. The totality of the possible combinations of inputs and outputs of the machine's logic circuits form the set of machine states. A gross functional check performed by exercising the computer's instruction set on a few operands can be seen to be less efficient and complete in verifying proper operation of all circuitry than comprehensive application of the set of inputs with comparison of resulting outputs against the set of unfailed machine output states. The one test method is superficial while the other is unnecessarily exhaustive. Each has been termed "100% testing" by industrial marketeers. The percentage of testing for this investigation will refer to the percentage of possible errors for which checking has been performed. The former method

mentioned above would probably yield a low percentage while the latter would represent testing in excess of 100%. The closer to the logic level that testing is directed, the more thorough testing becomes. Selective testing at the logic level can be most efficient in identifying all the failures of interest.

Not only must test procedures check for all possible failures of interest, they must also take care to avoid signalling error when none exists, as alluded to in Section III in the case of non-hardware-caused intermittent error. Testing which is not thorough leads to invalidation of the single failure assumption since some failures can go undetected. On the other hand, inappropriate error signals "crying wolf" can cause the pilot to take unnecessary abnormal action detrimental to mission completion. A significant advantage to testing conducted in the airborne environment is that not all errors identified airborne would be found if ground testing procedures were used instead. Consequently, ground maintenance personnel must have a high degree of confidence in airborne error indications since ground verification may be impossible. If a throwaway maintenance concept is in effect, good modules might be discarded because of inaccurate test results.

Detection of error is only one part of the test problem. Isolation of the causative failure is the other. Test procedures differ in their ability to provide fault isolation. Early test procedures were designed to produce isolation to the single component level (if isolation was provided at all) since machines were constructed with discrete technology. The multicomponent package of the higher level technologies has made unnecessary such fine resolution procedures. For the example BIT

design, the modular level is the level of interest. One is not concerned where within the module a failure is located; whether or not the modular package as an entity is faulty is of primary interest. With these general comments as a background, the various ways of categorizing test procedures can be explored.

B. PROCEDURE CATEGORIES

1. Normal vs. Marginal

Diagnosis of existing solid errors should be the first order of business for any test procedure. Prediction of possible future failures would be a desirable supplement to the preceding tests to locate existing errors. The former testing will be termed "normal" testing while the latter is called "marginal" testing. Normal testing will be the type pursued in the example test design. However, marginal testing conducted in conjunction with normal testing is generally valuable in furthering test objectives.

Intermittent errors cause one of the biggest problems to the test designer. However, an intermittent failure causing inconsistent results can often be forced to become a solid failure with a resultant solid error manifestation through marginal testing techniques [Ref. 7]. Marginal testing tends to worsen the third failure mode discussed in Section III by further weakening already deteriorated components until they become solid failures of the more easily diagnosed first failure mode. Marginal testing consists of overstressing components through the application of abnormal conditions to cause the weak ones to fail prematurely during test instead of later during normal operations. Stressing, for example, can consist of over or

under biasing transistors by a certain percentage of rated values. The danger of marginal testing is that existing intermittent failure can be masked by a rash of new failures should stressing be done carelessly or to needless extremes [Ref. 3]. When done carefully, however, marginal testing, in effect, predicts future failures by forcing them to occur at non-critical times. It also serves to identify and rid the machine of bothersome intermittent failure, thusly increasing the degree of confidence accorded to airborne test results.

Marginal testing is generally not appropriate airborne because of the time and extra equipment necessary to accomplish it. The accomplishment of marginal testing on the ground depends upon the maintenance concept. If periodic maintenance on the ground supplements airborne built-in testing, marginal testing should be part of this periodic procedure. In the example design, where no airborne repair is done, marginal testing can be accomplished whenever the computer is removed from the aircraft for repair of a solid failure identified by BIT.

2. Software vs. Hardware

Software testing refers to program-oriented, sequentially executed, periodic testing. The computer is directed by a program to accomplish a series of operations on supplied data. The results of these operations are then interpreted to provide diagnostic information. Since software testing is program-oriented, the level of testing (and therefore, to a certain extent, the efficiency of testing) is determined by the level of the programming language used. The lower the

order of the programming language, the closer to the component level operations can be specified. Assembly language or its equivalent is most frequently used.

A programmed test routine is sequentially executed, one instruction after the next. The length of the program in number of instructions, the cycle time of the storage device containing the program, and the execution times of the instructions affect the time duration of the test. Test results can usually only be determined after a sequence of instructions has been executed and a result determined. This result is then compared against some previously calculated correct result to see if error has occurred. The same sequence of instructions might then be repeated with a different set of data and a different expected result. Comparison against the norm can take place automatically under program control after short sequences have been executed, or later upon examination of a printed output.

Procedures for software testing differ widely. The detection and isolation functions can be accomplished concurrently or separately. In the separate case, an "executive" routine might be run periodically to determine in a gross sense whether or not the computer were exhibiting abnormal behavior. Once such behavior were sensed, a more detailed "diagnostic" routine might be run to determine the more exact location of the failure causing the error. Because of the limitations of the programming language in closely manipulating suspicious components, results might localize the failure to a region of the machine. Technicians would then locate the failure by hand probing. Such procedures tend to be inefficient, marginally effective, and always time-consuming.

The characteristics of software testing can be evaluated with regard to the BIT objectives. A definite advantage is that software testing requires little added hardware (other than storage) to accomplish the checking function. Isolation after detection is difficult because of the periodic nature of testing. The test program typically occupies core memory (unless slower peripherals are available for temporary storage) and requires significant running time if many different test data are to be used in an attempt to make testing more comprehensive. Some functional degradation would occur when time is scarce, even when the test program is run on a periodic basis, because testing must share available time with the operational flight program execution. On the other hand, the shorter the test program and the longer the interval between tests, the greater the danger of using erroneous results of undetected failure and downgrading test efficacy by invalidating the single failure assumption. Test results are only known after several operations have been executed. This presupposes that the machine has not failed to the extent that it cannot execute instructions and give results necessary to locate the failure. Intermittent failure would tend not to be detected by software testing, eliminating the problem of signalling error and indicating failure when none exists. On balance, software testing did not look generally attractive for the example design.

Hardware testing refers to checking accomplished by added circuitry. Such testing is characterized by simultaneous detection and isolation usually at the logic level, rapidly available results, and minimal degradation of operational capability. In general, the

added checking hardware generates a basis for comparison with concurrently generated flight program results, and actually accomplishes the comparison at the logic level. Operation at the logic level provides excellent fault isolation capability. Results of the comparison are known essentially immediately. If a fault exists, it can be located and appropriate action taken prior to contamination of other data, or utilization of erroneous results. Hardware testing differs from software testing in that it checks the correct operation of the circuit being tested, but does not verify the correctness of the data being operated upon. The effect is that each circuit in a chain must be so checked if resultant data is to be certified. Further discussion of concurrent testing, characteristic of hardware testing, will be presented in the next subsection.

By virtue of consisting of fewer components, checking circuitry is inherently more reliable as a whole than the logic it checks. However, the components themselves are just as subject to failure as the components they test. To provide a high confidence of valid testing, therefore, one must consider the added test hardware itself as a potential source of failure. Such hardware then becomes hard-core in the sense that its proper functioning must be verified before testing commences. Unlike the hard-core housekeeping and service hardware previously mentioned, checking hardware was considered to be part of the test problem.

Hardware testing offered many benefits making it attractive as a means of meeting the example design objectives within program constraints. Its obvious disadvantage relative to software test was

the much higher cost penalty incurred as a result of the expense of added hardware. A combination of hardware test to provide efficient test performance and software test to reduce expense offered a possible tradeoff for the example design.

3. Continuous vs. Periodic

Testing can be classified by its duration as either continuous or periodic. Continuous testing must also be concurrent (the results of test may be somewhat time-skewed) since ongoing operational computations occur simultaneously. Continuous testing is characteristic of hardware test. The effectively immediate failure detection provided by continuous testing tends to identify intermittent errors, where periodic testing does not. The single failure assumption is justified since failures are detected as soon as they occur. Operations can be halted upon occurrence of an error and the machine state at time of halt preserved. The process of "retry" or "restart" then attempts the last operation again to see if the same error recurs. Recurrence indicates a solid error and failure is flagged. Non-recurrence denotes an intermittent error, in which case the second correct attempt is used and operation continued. By noting the recurrence rate of intermittent error under the same conditions, intermittent hardware failure can often be distinguished from one-shot external sources. Hard-core housekeeping and service hardware is generally continuously tested.

Periodic test refers to checking conducted at specific intervals, such as software testing. The testing then time-shares with operational computation. Results are only determined after a number of sequential steps have been accomplished. Preservation of machine status for retry

when periodic testing detects an error is generally not practical. However, if the periodicity of test is sufficiently brief, error halt can occur shortly after failure, minimizing the cumulative effect of error on post-failure computation. The single failure assumption is still valid if the period between tests is short. Intermittent errors will not be detected by periodic testing until they become solid. Even in a continuously tested machine, hard-core checking circuitry is more reasonably tested periodically.

The unique nature of the added checking hardware providing continuous concurrent testing to the different logic circuits of the machine results in high cost. A tradeoff in favor of a periodic, interruptable test procedure exercised at frequent intervals appeared attractive for the example design.

4. Deterministic vs. Non-Deterministic

A deterministic test yields a definite answer to the question of whether or not an error exists. A non-deterministic test yields results which are interpreted statistically against an expected distribution to determine the probability of the existence of error. The terms are more often applied in relation to software testing procedures since hardware testing is always deterministic. Non-deterministic testing was not attractive for the example BIT design because of the requirement for a high degree of confidence in test results. Statistical techniques were, however, found useful in selecting initializing data.

5. Combinatorial vs. Sequential

Seshu and Freeman [Ref. 45] classify the organization of testing into two different categories, combinatorial and sequential.

A combinatorial testing procedure involves application of a fixed set of inputs to the machine with the output results being analyzed to identify failures. As an example, non-deterministic testing is combinatorial. A sequential procedure has no fixed set of tests which are applied. The result of the first test sequence determines which test sequence will be used next. Sequential testing is more efficient since selection leads to fewer tests. These two categories should not be confused with the often used classification of logic as combinatorial (combinational) or sequential. Combinatorial and sequential testing procedures clearly refer to classes of software testing and not to concurrent hardware test.

C. ALTERNATIVES

1. General

The previous section presented several categories which can be used to describe test procedures. In practice, the specific procedures presented in the literature tend to fall simultaneously into several of the categories previously mentioned; all are a blend of alternate approaches having favorable characteristics relative to their intended applications. The discussion of specific alternatives requires a further cataloging effort, difficult because of the diversity of approaches to test and because of the aforementioned overlapping of categories. The discussion presented is not intended to be comprehensive; it is meant to demonstrate the diversity existing in the test field and to introduce some techniques which proved useful in developing the specific blend of approaches best meeting the requirements of the example design.

Since most of the test alternatives identified have been presented in the literature, the discussions are usually short, rapidly settling to a single level of interest. Some discuss the systems approach, giving overall techniques for testing the computer's different major units. Others have developed schemes for determining the optimal test sequences for checking one unit of the computer (e.g., the arithmetic unit, or the memory). Such schemes examine the states of the elements comprising the unit under consideration, the elements being identified as either fault-free or failed, and develop tests to yield the final diagnostic results on the entire unit. Still other techniques examine the states of the inputs and outputs of a single logic element, or block of elements (e.g., an AND gate or a multiplier block), with the goal of locating a failed element. The presentation of alternatives below will generally move from the system level to the logic-block level; however, the typing is loosely defined and often difficult.

2. Coding

A large variety of schemes and a significant body of theory have been developed in the literature relative to coding test techniques. Generally, coding represents a succinct way of supplying redundant information to provide a norm for comparison. Codes can be used to detect and correct single or multiple errors. The program constraints imposed on the example BIT design eliminate from consideration error-correcting codes and those requiring core memory storage. For this reason, only parity was considered potentially applicable for the example design. Its nature and possible use will be discussed next.

Parity is the simplest error-detecting code consisting of one redundant bit of information, making the sum of the information bits plus the parity bit either even or odd as desired. For a binary number

$$N = a_1 a_2 a_3 \ldots . a_n$$

where a_i is the binary value for the ith bit location, parity P(N) can be expressed as

$$P_{even}(N) = \sum_{i=1}^{n} a_i \mod 2$$

and

$$P_{odd}(N) = \sum_{i=1}^{n} a_i + 1 \mod 2$$

The correct parity value for a data word is known a priori. Upon completion of an operation, the correct parity of the result is known and is generally attached to the result as an additional bit. The actual parity is then calculated and compared to the expected parity to determine whether or not error has occurred.

Parity has the capability of detecting odd numbers of errors, and therefore provides protection beyond the single error assumed. In the absence of the single error assumption, the risk of undetected multiple even errors can be calculated. Given an n bit word

$$N = a_1 a_2 a_3 \ldots . a_n$$

resulting from operations, the binary value a_i of the ith bit position can have one of two states relative to failure (failure states): it is either correct or erroneous. The probability of undetected error P_{ue} is just the sum of the probabilities of multiple even errors. Assuming an instantaneous probability of error p in bit location i and independence

between bit locations, the probability of k simultaneous errors is just p^k. Accounting for all combinations of ways k errors can occur in an n-bit word length (n even), the instantaneous probability of undetected error can be expressed as

$$P_{ue} = \binom{n}{2}p^2(1-p)^{n-2} + \binom{n}{4}p^4(1-p)^{n-4} + \ldots + \binom{n}{n}p^n(1-p)^0$$

$$= \sum_{k=1}^{m} \binom{n}{2k}p^{2k}(1-p)^{n-2k} \qquad \text{where } m = n/2$$

For $n = 24$ and $p = 10^{-3}$

$$P_{ue} \simeq 2.7 \times 10^{-4}, \text{ or } .027\%, \text{ a very low risk.}$$

Parity can be useful in both software and hardware test procedures. It is often used to detect single errors in data transmissions. For the example design its potential use was as a hardware test where the correct parity was automatically present, or generated by the circuitry to be checked. A hardware parity generator and comparator could then be added to provide error indication. An example application might be to a feedback shift register which always generates a number with odd parity to which a parity generator and comparator could be added to verify proper operation. The generation and use of parity for comparison was only acceptable for the example design where core memory storage of parity bits was not required.

3. Diagnostic Partitioning

The general technique of diagnostic partitioning divides the computer into smaller entities, each of which can then be tested separately. Forbes, Rutherford, and Steiglitz [Ref. 13] present such a technique in which the computer is partitioned into "diagnostic

subsystems," each having certain capabilities. The subsystem essentially is able to apply stimuli, sequentially execute a series of operations, receive and process inputs, and communicate diagnostic results of test to the outside world. The subsystems can then alternately diagnose each other. A sequence for system diagnosis at the subsystem level is developed. Their technique of partitioning a machine into essentially autonomous sections was found to be applicable in the example design. The test technique involves a periodic, software test with fault isolation provided by the order of operations. An interesting feature is the microprogramming of the test routine to provide closer manipulation of the logic for the reasons previously described in Section IV-A.

The concept of diagnostic partitioning can be applied to a partitionable machine in a "bootstrap" fashion. One subsection is considered to be hard-core, and it is checked by hardware means, manually, or by software. An example of software test would be execution of a small number of operations requiring only the hard-core subsection to implement. Upon verification of the hard-core subsection, one then uses it to check the next subsection. The two checked subsections can then be used to check the next and so forth. This represents a type of sequential testing (vice combinatorial) at the subsystem level. Manning [Refs. 31 and 32] describes a modification of such a technique. The difficulty with diagnostic partitioning is that the architectural designs of many computers do not facilitate partitioning.

4. Program Hierarchy Testing

A system technique related to diagnostic partitioning examines the functional capabilities of the computer. A hierarchy of distinct software programs is used to functionally partition the machine, in contrast to the physical sectioning associated with the diagnostic partitioning of the previous section. A high level program periodically functionally tests the computer by exercising short routines using the machine instructions to grossly check the computer for proper operation. Examples of functional checks might be adding, multiplying or shifting. Such "executive programs" are not intended to be comprehensive or isolating; they detect errors in functions by comparing results obtained to previously stored expected results. Once an error has been identified, a "diagnostic routine" tailored to the type of functional error detected is executed to provide the isolation required for repair. While not comprehensive, such a technique allows frequent running of the short executive routine, while calling on the longer diagnostic routine only when error is sensed. Cohen and Whitaker [Ref. 7] describe such a procedure developed at Sylvania. Bashkow, Friets, and Karson [Ref. 3] divide the diagnostic process by hierarchy into a command checkout phase, used to assure that the machine is "breathing" (no gross malfunctions exist), and "executive", "testing", and "diagnostic" phases to give more detailed checking at lower levels. The diagnostic programs used are microprogrammed to provide failure resolving capability.

5. Software Exercise, Hardware Detection

An interesting combination of testing techniques uses software routines to exercise the computer periodically and added hardware

circuitry to detect errors. The hardware provides the level of detection resolution required. Software routines need only thoroughly exercise the machine, with no attention to order of execution for isolation being necessary. Fred Lee [Ref. 27] describes such a procedure in which the machine's operations are broken down into sequences of events, recognizable as pulses occurring in a specific order. The correct sequence is provided for the test routine and is compared against the actual sequence. Hardware monitoring devices provide the comparative function with non-coincidence signalling specific error. With an 18.2% increase in transistor count for test purposes, Lee claims 100% confidence in the device. This procedure is also described by Sellers, Hsiao and Bearnson [Ref. 43] under the title of "sequential logic latch checking." While Lee's procedure was not used, the idea of software exercising and hardware detection was of use for the example design.

6. The Black-Box Approach

The black-box approach refers to the process of setting the inputs of a network and observing the resultant outputs, useful information thereby being derived without internal access to the network. A most extensive body of literature reports on varying schemes to obtain optimal, minimal sets of inputs to diagnose all possible errors internal to the network. With the growing use of multicomponent packages inaccessible internally (IC, MSI, and LSI technology), this test area has received renewed attention. Eldred [Ref. 12], in one of the earlier papers treating the black-box approach, discussed the derivation of minimal tests for a simple

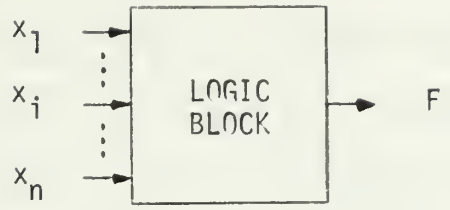
network of discrete components by evaluating the input conditions which should cause the network output to be "activated" or "inhibited." Results deviating from this norm indicated failure. Armstrong [Ref. 1] presented a procedure based on "path sensitizing" in which a given internal fault is selected and its effect is traced to the output for given input conditions. The procedure continues until all faults have been treated and the significant input and output patterns derived. The "truth table" or fault dictionary technique is similar in that a table of the expected outputs for given inputs and specified internal failures is derived. Comparison of combinatorial test results to the fault dictionary determines if an error has occurred, and where.

The derivation procedure for a large block of logic can be tedious, even when computer aid is used. The requirements for memory can easily exceed availability in the analysis of large networks. Such difficulties have led to the development of simplifying methods for automating the analysis of large networks. There is wide agreement in the literature that the derivation of minimal input tests for a large block of logic must be automated.

Sellers, Hsiao and Bearnson [Ref. 42] developed an algebraic technique based on Boolean difference to facilitate learning the effect of a change in state of a chosen input on the network output. The procedure involves logically Exclusive-ORing the Boolean output function, expressed in terms of the inputs, with the same function having the chosen input inverted. If the Boolean output function is

$$F(x_1, x_2, \ldots, x_i, \ldots, x_n)$$

where x_i are the inputs, for the system



they define the Boolean difference as

$$F(x_1, x_2, \ldots, x_i, \ldots, x_n) \forall F(x_1, x_2, \ldots, \bar{x}_i, \ldots, x_n)$$

where the chosen input of interest x_i is inverted in the second expression and $\forall$ represents the Exclusive-OR operator. The Boolean difference yields the input conditions for which the output will change state, given the chosen input state change.

Roth [Ref. 41] with his calculus of D-cubes expands on the above method, but with a more graphical technique to solve the sometimes formidable problem of accomplishing algebraic operations such as $\forall$ for complex functions. He first expresses the truth table of each element of the network in a succinct form and then gives rules for intersecting the tables of the individual elements to form the table describing the entire network.

The usefulness of such techniques is reported by Galey, Norby and Roth [Ref. 14] in an earlier version of Roth's later technique. Four eight-bit input tests were automatically derived, the results of which would indicate whether any one of 102 possible internal failures had occurred (but not which one). This illustrates the concept of testing an internally inaccessible network for failure without interest in which specific component has failed.

An interesting contrast is offered by Maling and Allen [Ref. 30] who test a network for failure with the purpose of identifying the specific failed component. For each n-input component of the logic net, 2^n represents the number of different input combinations. Only n + 1 of these are necessary to show that each input in turn can control the output and that the output can take either state. For a net of k such components where the ith component has n_i inputs, they state that the number of configurations C of the n + 1 required inputs per component is

$$C = k + \sum_{i=1}^{k} n_i$$

This number also represents the maximum number of tests required to thoroughly check the circuit with component isolation. The lower bound is determined if each test is efficient enough to eliminate half the components from further consideration. The minimum number of tests T_{min} is then

$$T_{min} = 1 + |\log_2 C|$$

where | | indicates next higher integer. From experience, they state that the number of tests required is usually approximately equal to the number of components.

7. Non-Duplicative Hardware Checking

Checking by adding hardware which does not duplicate the circuitry being checked provides the benefits of hardware test without the cost of duplication. Rao [Ref. 39] describes a method for checking arithmetic-type operations in a processor through the use of residue coding generated and employed by added hardware without storage to

identify errors but not to locate them. The residue code was used to provide a high level of multiple-error checking capability not required in the example design. The 1000 gate processor required 400 added gates to check it, or a 40% increase in cost which would be unacceptably high for the example design. Sellers, Hsiao and Bearnson have compiled a comprehensive volume [Ref. 43] on error detecting logic, which is the only one of its kind identified by the author. The cited reference is an excellent source of non-duplicative hardware checking schemes. The use of non-duplicative hardware schemes appeared attractive for the example design, particularly for the hard-core circuitry included in the test problem.

8. Replication and Comparison

When other schemes do not provide adequate checking, one can replicate circuitry, operate the replicated portions in parallel and compare the results, with any non-coincidence indicating error. While the technique is expensive (and unacceptable for the example design) when employed on a large scale, it often presents the only technique by which isolated small blocks of circuitry, or highly irregular circuitry can be thoroughly checked. For the example design, duplication of small sections was very useful. The replicate and compare concept is often applied when high reliability requirements force the use of redundant hardware on a large scale. Switching to the unfailed duplicate offers continued operation while the failed portion is repaired. Automatic repair is not appropriate to this investigation, yet it proceeds naturally from some of the methods found useful and therefore represents a good topic for further related investigation

An interesting contrast is offered by Maling and Allen [Ref. 30] who test a network for failure with the purpose of identifying the specific failed component. For each n-input component of the logic net, 2^n represents the number of different input combinations. Only n + 1 of these are necessary to show that each input in turn can control the output and that the output can take either state. For a net of k such components where the ith component has n_i inputs, they state that the number of configurations C of the n + 1 required inputs per component is

$$C = k + \sum_{i=1}^{k} n_i$$

This number also represents the maximum number of tests required to thoroughly check the circuit with component isolation. The lower bound is determined if each test is efficient enough to eliminate half the components from further consideration. The minimum number of tests T_{min} is then

$$T_{min} = 1 + |\log_2 C|$$

where | | indicates next higher integer. From experience, they state that the number of tests required is usually approximately equal to the number of components.

7. Non-Duplicative Hardware Checking

Checking by adding hardware which does not duplicate the circuitry being checked provides the benefits of hardware test without the cost of duplication. Rao [Ref. 39] describes a method for checking arithmetic-type operations in a processor through the use of residue coding generated and employed by added hardware without storage to

identify errors but not to locate them. The residue code was used to provide a high level of multiple-error checking capability not required in the example design. The 1000 gate processor required 400 added gates to check it, or a 40% increase in cost which would be unacceptably high for the example design. Sellers, Hsiao and Bearnson have compiled a comprehensive volume [Ref. 43] on error detecting logic, which is the only one of its kind identified by the author. The cited reference is an excellent source of non-duplicative hardware checking schemes. The use of non-duplicative hardware schemes appeared attractive for the example design, particularly for the hard-core circuitry included in the test problem.

8. Replication and Comparison

When other schemes do not provide adequate checking, one can replicate circuitry, operate the replicated portions in parallel and compare the results, with any non-coincidence indicating error. While the technique is expensive (and unacceptable for the example design) when employed on a large scale, it often presents the only technique by which isolated small blocks of circuitry, or highly irregular circuitry can be thoroughly checked. For the example design, duplication of small sections was very useful. The replicate and compare concept is often applied when high reliability requirements force the use of redundant hardware on a large scale. Switching to the unfailed duplicate offers continued operation while the failed portion is repaired. Automatic repair is not appropriate to this investigation, yet it proceeds naturally from some of the methods found useful and therefore represents a good topic for further related investigation

by others. Duplication and comparison, recognized as one of the most effective test techniques, formed the basis for a unique application in the example design of the diagnostic partitioning scheme described earlier.

9. Probabalistic Method

A non-deterministic method which is periodic and combinatorial is presented by Merwin [Ref. 33]. A block of combinatorial logic (vice sequential logic having feedback paths, not to be confused with combinatorial test) having many inputs is tested by first establishing the expected distribution of output values. Each of the possible combinations of input values is considered equally likely. The output pattern resulting from each input pattern is derived. The statistical appearance of a given logical value at each specific output of the output set can then be determined. For example, if there are 16 possible input combinations (four inputs) and three outputs, output number two may have the value logical one for eight of the input combinations. The logical value one would then be expected 8/16 or 1/2 of the time at output number two. Merwin attaches a random number generator to the inputs and tabulates the incidence of appearance of the logical value one at each of the outputs. Deviation of the actual ratios from the expected ratios may signify an error. If output two took the value logical one only 1/16 of the the time instead of the expected 1/2 of the time, error would be likely. Decision criteria can be established using statistical procedures. The random number generator as a source of random bit patterns was useful in the example design.

V. THE EXAMPLE DESIGN

A. THE TEST CONCEPT

The parent computer was divided into units:

1. The processor unit - containing arithmetic logic and general purpose registers.
2. The control unit - to provide control signals for direction of operations in the processor unit.
3. The core memory unit - to provide storage of the flight program and temporary data.
4. The input/output (I/O) unit - to provide interface between the computer and the equipment it serves.

The I/O unit will not be considered in the present investigation.

The proposed instruction set for the computer (to be termed the macro-instruction set) provided for an extensive half-word/ half-register addressing and manipulation capability. Processing was to be possible on 24-bit words (full-word operations), on the right or left 12-bits of the 24-bit word separately (separate half-word operations), or on the right and left 12-bits of the 24-bit word simultaneously (parallel half-word operations). With little added hardware and design effort, it appeared possible to configure the highly regular logic of the processor unit into two autonomous halves, each possessing multi-functional capabilities. This diagnostic partitioning in effect provided a duplex redundant processor unit without the expense of duplicating the hardware. This technique will be termed "split duplication."

With the proposed high speed of the parent computer, sufficient time was available when the machine was not performing its basic

operational mission (idle time) to time-share a periodically exercised test procedure without imposing any functional degradation. This would be particularly true if the test procedure, once initiated, could be interrupted to return the computer to operational computation without destroying test efficacy. Idle time was to be available every few seconds, validating the single failure assumption through short periodicity of test. The lower cost advantage of periodic, program-oriented testing could be thereby enjoyed.

Two modes of operation were identified. In "normal" mode operation, denoting mission operational computations, both halves of the computer would be used together, making full-word, separate half-word, or parallel half-word operations possible. In "test" mode, denoting idle-time test exercising, only parallel half-word operations would be possible. During test mode, the autonomous processor halves would be loaded with identical half-word bit patterns. Identical parallel operations would then be executed on the like data independently. Comparison of the results would then be accomplished with non-coincidence of the two halves indicating error. The advantages of the superior duplication and compare method could be enjoyed without the cost disadvantage of duplicated hardware.

The source of data words with which to initialize the two processor halves during test mode remained to be resolved since core storage was not acceptable. The possibility of using an inexpensive hardware pseudo-random number generator, similar to the one used in Merwin's probabilistic method, appeared to be an attractive option which was compatible with the concept of interruptable test while requiring no

core storage. Random patterns would more nearly simulate inputs used during normal mode operation. An argument can be made for "worst-case" testing in which a small number of unusual bit patterns not normally encountered in normal mode operation are used to stress the machine in a worst-case manner. Such stressing appeared to be more appropriate for marginal testing on the ground when such worst-case patterns might be expected to hasten impending failure. Additionally, no "end-of-test" point needed to be identified since the machine was to revert to test mode at any time not required for normal mode operation. Finally, the storage required for worst-case bit patterns obviated their further consideration.

The use of a pseudo-random number generator allowed the core memory unit to be disconnected from the processor unit during test mode, and made possible the core memory unit's separate checking either concurrently, prior to, or subsequent to processor unit test. The control unit, however, was required in test mode to supply the control signals to direct the parallel half-word operations. Testing of the control unit itself, and the location and execution of the exercising test routine still needed resolution.

The issuance of accurate control signals by the control unit to the processor unit is a prerequisite to correct computation. The control unit was to be microprogrammed using a read-only-memory (ROM) as the storage device. The control signals appropriate for executing the macro-instruction set were to be hard-wired in the form of short routines of the lower order micro-instructions. The hard-wiring consisted of arrays of transistors implemented on a small number of

silicon chips, the whole comprising the ROM. The remainder of the control unit consisted of the selection and sequencing circuitry required to assure issuance of the proper signals in a timely manner.

Because of the standard packaged arrays available with which to implement the ROM (the low risk nature of the program dictated use of off-the-shelf hardware), sufficient unusued storage capacity beyond the requirements for the microprogrammed control signals was present to allow storage of a microprogrammed test routine. Careful, efficient micro-programming of the test sequences promised a much shorter test routine requiring significantly less ROM storage than the comparable core memory storage needed for an equivalent routine programmed using the macro-instruction set. The inherent advantage of the lower order micro-instruction set relative to thorough exercise of the computer at the logic level is enjoyed by such a scheme. An additional significant advantage for an interruptable, time-shared test routine is the much shorter cycle time of the read-only-memory compared to the core memory.[9] Note should be made here that test mode exercise of the processor unit could be accomplished entirely independent of the core memory unit.

Since the control unit was to issue the control signals directing the test routine, it became hard-core hardware whose proper functioning had to be continuously assured. Hardware techniques for continuous, concurrent testing of the control unit were therefore essential to the

---

[9] A typical core memory cycle time is 2 μsec while a typical ROM cycle time is 200 nsec, 10 times faster.

concurrent testing of the control unit were therefore essential to the test concept. As will become evident when the control unit BIT design is discussed, the highly irregular nature of control unit circuitry tends to necessitate hardware test techniques in any case.

With the test concept developed, the more detailed BIT design of each unit can now be examined.

B. THE PROCESSOR UNIT

With the exception of the power supply, considered to be hard-core servicing hardware excluded from the test problem, no hard-core hardware requiring continuous test was to be located in the processor unit. The split duplication, periodic technique of testing the processor unit could be expected to thoroughly check its operation.

The contents of the general processor module resulting from partitioning the processor unit are shown in Figure 2. Figure 3a shows the 24-bit data path divided into four-bit groups, with the double line denoting the left and right half-word division. Two four-bit groups L_i and R_i, are physically located on the same module, providing eight bits of the 24-bit wide data path. The remaining groups are likewise associated on separate modules, a total of three identical modules (see Figure 2) being necessary to implement a 24-bit path. Modification of word length in eight bit increments is possible, in consonance with the objective of flexibility of word length. For example, addition of a fourth identical module would easily convert the processor to a 32-bit path width.

Emphasis should be placed on the fact that the description above refers to a data path, and not to a single register or a single

functional circuit. The amount of hardware implemented on the 2A NAFI module is dictated by its area and pin limitations, discussed in Section II-B-2. The entire processor unit can then be thought of as consisting of a series of three-module sets, the modules within each set being identical. The total number of modules in the processor unit would be a multiple of three.

Providing isolation to the modular level has only been briefly discussed so far. Identical half-word bit patterns are used to initialize the processor circuitry being tested. While the computer is in test mode, these bit patterns undergo parallel operations concurrently in the autonomous halves. The results of such operations should therefore be identical at each point in the data path. Any difference indicates that a fault exists. Non-coincidence is signalled by a hardware comparator placed in each module to compare the autonomous halves' results. The required fault detection and isolation are hence achieved by the placement of the comparators in the data path at the modular level. Comparison takes place continuously during test mode at each clock pulse, so interruption to return to normal mode operation has no effect on test efficacy.

The decentralized power supply located in each module consisted of the final step of regulation required to provide the power level or levels necessary in the module. The decoding of the control signals was also accomplished in the associated module. Decode could thereby be checked by the same technique as other processor hardware, eliminating the necessity for the more difficult, costly continuous checking of decode circuitry located in the control unit. Any failure in

the power supply serving the module or in the decoding function would occur in the module. By tying the hard-core checking circuitry for testing the local power supply (not treated herein) into the processor module checking circuitry, a single error signal could be issued from the module in case of failure. For the example design, the reason for failure within the module did not need to be identified; only isolation to the modular level was required. If a centralized power supply provided fine power regulation and if decode were located outside the module served, precautions would be necessary to insure that failures in these functions did not cause failure within the module to be erroneously signalled. Confidence in the error signal once issued is increased by the decentralizing scheme described.

In test mode, only parallel half-word operations are accomplished. In normal mode, however, full-word and separate half-word operations are also utilized. Differences in the execution of operations in the two modes had to be identified to ensure that test procedures thoroughly exercised the circuitry, and that test hardware did not degrade normal mode operation. The carry forward found in adders, shift registers and counters in the processor unit was the major such difference.

Figures 3b and 3c show the carries associated with parallel half-word and with full-word operations, respectively. In the case of parallel half-word operations, the carries between adjacent four-bit groups in the two halves are identical. For example, the carries from L_1 to L_2 and from R_1 to R_2 are the same. Since the L_1 and R_1 groups are located in the same module, the carries from the most significant ends

of L_1 and R_1 are identical when no fault exists. These carries can then be compared, with non-coincidence indicating a failure in that module.

One difficulty arises during test mode parallel half-word operations when an error in a carry is detected; e.g., the carry from L_1 to L_2 differs from the one from R_1 to R_2. Error is signalled in the current module. The differing carries, however, cause the bit contents of L_2 and R_2 in the next module to differ, and because they don't compare, error is also signalled in the next module. This difficulty can be resolved by inhibiting the error signal in module i+1 when an error signal is issued from module i preceding it.

Another difficulty arises because during full-word operations in normal mode, the bit contents of the groups L_i and R_i in the same module may differ with no faults existing. Likewise, the carries propagated from L_i to L_{i+1}, and from R_i to R_{i+1}, may also differ. The error signal due to non-coincidence must only be allowed in test mode, in which any non-coincidence is the result of failure. A test-enable signal can be applied to checking circuitry in test mode.

It was also desirable to eliminate any gating from the inter-modular carry paths to avoid propagation delays. Figure 4a shows the checker circuitry added to each module. Figures 4b and 4c show possible logic implementations of the desired truth tables for the carry checker and error-inhibit respectively. Figure 5 shows the relationships between two adjacent modules.

Note should be made that the error inhibition in the case of the first difficulty discussed does not allow two adjacent modules to signal

error during the same periodic test iteration. Both carries from one module to another are also assumed not to fail simultaneously, in which case the comparison check would be passed in spite of existing failures. Both of these cases are highly improbable and represent part of the undetected failure risk accepted under the single-failure assumption for built-in-test. In the case of simultaneous failures in adjacent modules, only one is signalled. However, upon checkout after repair or replacement of the signalling module, the second module would then immediately indicate failure.

The fault-detecting circuitry described thus far does not distinguish between faults occurring in the module and faults occurring in the data transfer paths between that module and the previous one. Circuitry to provide such isolation could be added, and would consist of another comparator if the additional cost were acceptable. The problem of determining if the fault exists in the module or in the data transfer paths between that module and the preceding one would have to be accomplished by ground maintenance personnel unless the additional comparator were incorporated.

The pseudo-random number generator has been very briefly treated. Such a device is capable of providing long sequences of data words. A 12-bit generator was required for the example design. Golomb [Ref. 15] describes the design of a simple linear feedback shift register requiring very little hardware. An example generator which adequately fulfills the test requirements under consideration is included as Appendix A. The maximum length sequence of 2^{12} different patterns was obtained by implementing a modulo two irreducible polynomial found in Peterson

[Ref. 36] and adding the nonlinearity of the important all-zero case. The patterns so obtained met Golomb's tests of randomness in each bit location. A self-checking pseudo-random number generator design using more hardware is illustrated by Sellers, Hsiao, and Bearnson [Ref. 43] under the title of "unit distance code parity checked counters."

C. THE CONTROL UNIT

The control unit was the least regular of the units to be self-tested. Additionally, it was hard-core, requiring continuous test to validate the control signals issued to the processor from the ROM. The split duplication test concept of periodically exercising the processor unit during idle time presupposed a fault-free control unit able to issue appropriate control signals to direct test exercises whenever such idle time became available. Continuous testing of the control unit with added checking hardware would assure its fault-free availability by signalling its unavailability upon occurrence of a failure. Partitioning the control unit to provide modular isolation of failure while minimizing the requirement for added hardware is the subject of this section. Since the control unit was the only unit requiring continuous test, it should be recognized that a large portion of the overall hardware penalty for providing BIT to the computer as a whole was to be paid in the control unit.

Testing the control unit consisted of the following steps:

1. Testing the ROM for correct word content
2. Testing proper accessing of the ROM
3. Testing proper sequencing of accesses
4. Testing the checking hardware, which was also subject to failure.

Testing the checking hardware was a problem common to all the units, and it will be treated in Section E below. Figure 6 shows the non-partitioned control unit organization for the parent computer. Figure 7 illustrates the general modular partitioning and hardware added for checking, which is described below.

Testing the ROM for proper word content will be examined first. The control signals used to properly execute the flight program (and the test routine) are stored in the ROM in the form of hard-wired bit patterns called microwords.[10] The contents of the microword can change under failure, having a catastrophic effect on the control unit's ability to issue proper signals and consequently on the computer's ability to execute the flight program. The ROM, exclusive of addressing hardware, will be assumed to be implemented in segments of 256 eight-bit words, shown in Figure 8, although this implementation is not critical to the test procedures described. The ROM microword length will be assumed to be 48 bits, also not critical. The ROM is then implemented in six segments, as illustrated in Figure 8.

Three fields of the microword format (see Figure 8) have test significance:

1. Parity field (P) - one bit dedicated to parity of the entire microword in which it is located.
2. Next address field (NA) - eight bits containing the next address in the microprogram sequence (the next microword to be executed). This field was necessary even without BIT.

---

[10]Depending on the method of microprogramming, a microword may include several micro-instructions to control simultaneous operations in the processor and elsewhere. A microprogram is executed one microword at a time.

3. Current address field (CA) - eight bits containing the address of the microword in which it is located.

The six segments comprising the ROM are checked for correct word content by parity. The addressing circuitry accesses only one microword at a time. Parity is generated on the microword issued to the 48-bit hold register. This generated parity is then compared to the proper parity stored in the parity field of the microword. Note should be made that the hold register and the ROM sense amplifiers are also checked by this procedure. The functions of parity generation and comparison are combined in the parity checker shown in Figure 7.

The partitioning indicated shows all addressing and decoding circuitry in a module separate from the ROM storage segments, sense amplifiers and hold register. Divorcing the circuitry functionally related to addressing in this manner allows fault isolation to the modular level. This technique eliminates the ambiguity as to the modular location of failure when a portion of the addressing function is implemented in the same module as the ROM storage segments (a good example is the address decode, often provided on the same MSI chip as the storage devices).

The single-failure assumption made for the example design contends that the probability of multiple simultaneous failures in systems composed of components having inherent high component reliability is so small that practical test design need not consider it. This assumption was justified for discrete components and even for IC's, but with the advent of MSI and LSI with their numerous closely-packed components, it must be reconsidered. In the context of the present subject, one must consider the higher probability of multiple failure

caused, for example, by a cracked silicon chip where several adjacent components would be simultaneously affected. Odd parity, for instance, will not detect multiple even failures. The use of parity for ROM content checking appears to be justified by the fact that multiple failures would tend to affect more than one microword (to continue the example, a chip crack probably would not lie straight along the line of devices implementing a single microword). While one ROM access might not catch an even number of failures in one microword, very few subsequent accesses to different microwords would be necessary before a single or multiple odd failure would be detected and signalled. So, while the single failure assumption can be questioned for an MSI ROM implementation, the use of parity can still be justified.

Testing the addressing functions of the ROM is accomplished by comparing the current address field (CA) of the microword with the step counter contents. The step counter (or a second register if timing requires the step counter to change prior to the issuance of the microword being accessed) contains the address of the microword to which access is being attempted. The eight bits of the CA field contain the address actually accessed. Comparison of the two indicates whether an addressing failure has occurred. The step counter, decode and drivers are implemented on the same module. Non-comparison of the CA field and the step counter therefore signals an error in this address-function module. If the parity check in the ROM storage module fails, indicating incorrect microword content or a failed hold register, the error signal from the address function module is inhibited since the contents of the CA field being used for comparison are now suspect.

Proper sequencing of accesses to the ROM is the most difficult check to accomplish. A description of the sequencing process in general terms gives insight to the problem. The microprogram contained in the ROM consists, in effect, of a series of "subroutines" in a lower level language (the micro-instruction set), one "subroutine" for each of the macro-instructions used to write the flight program stored in the core memory. The flight program instruction word's operation code field, representing the macro-instruction, is analogously used as the "call" statement for its "subroutine". Since the same micro-instructions may be used in different mix to implement different macro-instructions, the number of micro-instructions is, in general, smaller than the number of macro-instructions.

Given a new flight program instruction word to be executed, the first access to the ROM is dictated by the operation code field of the instruction. This operation code is decoded as a selection of one microword in the ROM. Subsequent accesses to the ROM until the "subroutine" started by the operation code "call statement" is completed are dictated by the NA field of the microword itself. At the end of the sequence, the microword indicates that the sequence is complete and a new flight program instruction word is fetched by the FETCH CONTROL. Under certain conditions (such as repeats and branches), the repeat counter and condition code register dictate that the NA field be ignored and that the step counter (ROM address register) be incremented or decremented to indicate the next ROM address to be accessed. There are, then, several different sources of the next ROM address to be accessed:

1. The operations code field of the program instruction word found in the instruction register (U_1, U_2, U_3 in Figure 6) dictates the initial access to the ROM in executing a given program instruction word.
2. The NA field of the microword just accessed indicates the next ROM address to be accessed except that:
3. The repeat counter and condition code register can dictate direct modification of the step counter to yield the next ROM address to be accessed, in which case the NA field of the last microword accessed is ignored.

The SEQUENCE CONTROL selects the proper source of the next ROM address to be accessed. It modifies the step counter as required by the repeat counter or condition code register, and selects the proper field (U_1, U_2, or U_3) from the instruction register dependent on whether half or full-word instructions are being executed. When the NA field is selected as the source of the next address, its contents could be held in a separate register until they could be compared with the CA field of the microword actually accessed to see if a proper accessing had occurred. However, because of the possible other sources of the next address, it appeared that the proper functioning of the SEQUENCE CONTROL, FETCH CONTROL, REPEAT COUNTER, and CONDITION CODE REGISTER could only be assured by duplication, parallel operation, and comparison for identical results. Only in this way did adequate continuous checking of the proper sequencing to accesses seem feasible.

While the duplication and comparison test method should be reserved for last consideration, as indicated in Section IV-C-8, its application to the small logic sections described here appeared to be required to provide continuous checking. Controls which are duplicated and compared can be placed in any module as long as the duplex circuitry and comparator are in the same module. Partitioning of this duplicated

circuitry was therefore dependent on 2A NAFI module limitations only. The portion of Figure 7 labeled SEQUENCE CONTROL MODULE, then, could be broken into several modules with isolation of faults to the modular level still provided.

D. THE CORE MEMORY UNIT

Modification of an existing design to meet the requirements for a 24-bit word length, 8K core memory for the parent computer was considered. The use of an already developed memory design appeared favorable in light of the short schedule and low risk nature of the program. Although the final choice of memory type and size was dependent on changing requirements and therefore not firm, the example design will consider modifications of the basic design shown in Figure 9 to provide a BIT capability with fault detection and isolation to the modular level as the goal. The memory to be modified, termed the "standard memory unit" (SMU), was a 3D, coincident current, 32-bit word length, random access, 4K core memory. The example used serves well to demonstrate the factors involved in memory test.

Reference 35 briefly summarizes the standard techniques for functionally exercising a core memory. The functional exercisers listed below check for proper operation of the memory as a black-box without examining specific internal circuits. The standard functional exercisers are:

1. Check-sum - checks proper memory loading. This check can be accomplished using the flight program and constants stored in the computer for the mission.

2. One's discrimination - checks memory's ability to read and write ones coreectly. Memory buffer registers, sense amplifiers, the core array, and driving circuits are checked by this test.
3. Zero's discrimination - checks the memory's ability to read and write zeros correctly. The driving circuits are checked by this test, as well as the sense amplifiers' sensitivity to noise.
4. Addressing - checks whether or not each memory location can be correctly accessed. In addition to those circuits tested by the discrimination tests, the memory selection logic, decoders and drivers are checked.
5. Checkerboard and Inverted Checkerboard - these tests produce worst case noise conditions upon half-ready, which results in maximum inhibit noise whenever a zero is written. The inhibit noise from a cycle where zero was written can cause an error during the read portion of the next cycle.

The discrimination and checkerboard tests are aimed at discovering marginal conditions, and were not considered appropriate for airborne testing. They would certainly be appropriate as part of pre- or post-flight checkout on the ground, as discussed earlier in relation to marginal testing in general. The check-sum and addressing tests, more suited to discovering existing solid failure, appeared to be appropriate for in-flight application.

The five tests enumerated above are program-oriented, periodically exercised tests. Test techniques which require added hardware include coding and separate checking circuitry for each circuit type. Coding, principally parity, is popular for checking memories, but this technique fell outside the program constraints for the example design. Techniques for adding specialized circuitry to test the memory are described in Ch. 14 of Ref. 43. The additional expense of the circuitry and complete memory reconfiguration appeared inappropriate for the design modification intended.

Modification of the modular partitioning of the SMU appeared necessary to facilitate the test design if isolation to the modular level were to be accomplished. While packaged employing standard NAFI modules, the SMU did not use the 2A size, but rather the 1A and 1B sizes.[11] The standard memory unit was implemented with the equivalent of 152 1A NAFI modules. As evident in Figure 9, partitioning was done by circuitry type; e.g., there are 16 1B size sense/inhibit modules, one 1A address register module, and so forth. Several modules, of different types, are involved in one memory access; an address register module, address decoder module, timing control and timing modules, and sense/inhibit modules are all involved in one access. It is difficult to determine airborne in which module the fault lies once one is detected by a functional test alone. A unique way of applying functional tests and some added hardware were required to accomplish the modular isolation capability required.

Sixteen 1B NAFI modules were used in the SMU to implement the sense/inhibit functions for the 32 memory planes (32-bit word length). This represents circuitry for two planes (bit locations) per 1B module. An estimate (based on area limitation because of an essentially discrete component implementation of sense/inhibit circuitry, and allowing for added checking hardware) indicated that eight 2A

---

[11] The number in the NAFI size designator refers to the horizontal dimension of area (width), while the letter refers to the thickness. 1A is the smallest basic size, having unit standard width and unit standard thickness. The 2A module is twice as wide as the 1A and hence has twice the area, and the 1B is twice as thick as the 1A [Ref. 10].

modules should amply suffice to implement the sense/inhibit functions for a 24-bit word length. Three planes were to be served per 2A module. It was envisioned that bit locations served by a module would be adjacent. Figure 10 illustrates the scheme.

The implementation of the decode function for the SMU required two modules dedicated to X select and two to Y select, each module serving the entire core stack. An approach to partitioning which initially appeared attractive was to partition the decode logic so that the X and Y decode serving a smaller block of the core stack would be placed in the same module. However, partitioning the decode in effect doubles the logic required for every partitioning (e.g., placing the X and Y decode for one quarter of the core stack in one module would, for the entire core stack, entail quadruplicating the logic). Duplication and comparison required only twice as much decode logic, and this method was chosen. For example, the circuitry on one of the two X decode 1B modules is duplicated, the duplex hardware being placed in the same 2A module. Figure 11 shows a decode module. Four 2A modules were required for decode in the example design.

The address register also required duplication for separate test by the duplication and comparison technique. Checking of power supplies, transient protection, temperature tracking voltage sensors, timing, and associated regulators have been excluded from consideration, as they are hard-core housekeeping and service functions. The major areas subject to failure during flight are the decoding, sense/inhibit and select lines, cores, drivers, and amplifiers associated with accessing the memory, which are checked by the procedures described herein.

Fault isolation to the functional module level plus the core stack is provided by the test procedures described below. Faults occurring in the sense/inhibit functions are isolated to a single sense/inhibit module and core stack combination. Faults occurring in the addressing function are isolated to a single address register module or decode module. Faults occurring in the core stack are isolated to the core stack only if all tests can be conducted. No airborne discrimination between a single sense/inhibit module and the core stack appeared feasible if the sense/inhibit test failed because later tests could not then be confidently conducted. Such discrimination is easily accomplished on the ground. While a higher degree of isolation would be preferable, the level provided airborne closely focuses the efforts of maintenance personnel and greatly reduces the time/cost of maintenance. Sub core-stack isolation would probably not be useful since the core stack must be treated as an entity by maintenance personnel.

Testing of the sense/inhibit functions should precede testing of the decode function to insure that the latter tests are valid when conducted. The sense/inhibit functions serve the entire core stack; that is, a single sense amplifier & a single inhibit driver serve the same bit location in all the 8K words of the core stack. Each access to the core memory exercises all the sense/inhibit circuitry since all the bit locations of the word are involved. Solid failures result in a stuck-at-one or stuck-at-zero condition in a bit location. To isolate such fault manifestations to the sense/inhibit module or the core stack serving the bit location, one must first detect the fault and then relate it to the proper module. The test consists of attempting to access a core memory location which contains a previously

stored constant, A location containing all one's tests for the stuck-at-zero condition. Another location containing all zero's tests for the stuck-at-one condition. Two core memory locations are therefore dedicated for test use, one containing all one's and the other all zero's. A second set of such tests using the same cells should be performed to verify the restore operation; however, discrimination between failures in the sense/inhibit module and the core stack would still not be provided because of the possibility of a broken sense line (which also looks like a sense amplifier stuck-at-zero). Relating the failure to a specific sense/inhibit module is accomplished by checking hardware added to each module. Assuming eight sense/inhibit modules with three-bit locations served per module (24-bit word), one adds a three-bit register to each module (that is, in effect, a partitioned output buffer register for the core memory). A three-bit comparator (XOR) senses the failed condition when the three-bit locations are not identical. For example, stuck-at-one failure in the fourth bit location would be detected by accessing the memory location containing all zero's. The three-bit register of the second sense/inhibit module (serving the second three-bit group of the 24-bit word) would read 1 0 0 , producing an error signal from the XOR circuit on the module. Figure 12 shows the configuration of the sense/inhibit module.

The exercising procedure for the decode function and the core stack consists of check-summing over sections of memory. The core memory contains the stored program and constants (unalterable part of memory) which cannot change during flights, and a small section (scratch pad) reserved for storage of data which can change in-flight.

Scratch pad test will be discussed separately. Check-summing is accomplished by cumulatively adding the contents of all the cells of the unalterable part of memory modulo 24, the final sum accruing in the accumulator.[12] The expected check-sum (ECS) for the unalterable part of memory has been previously calculated externally and stored in the memory as a constant. Coincidence of the calculated sum and the ECS (subtraction is often used to give an expected zero result) indicates not only that the program stored in that part of memory is intact, but also that the accessing process has been properly accomplished.

Sequential access to each cell of the segment is attempted during calculation of the sum; the sum will check with the ECS only if every access has been properly executed. The accessing process thoroughly exercises the core stack and its associated decode modules. Isolation of faults to the decode module (by its internal comparator) or to the core stack (by an incorrect check-sum) is thereby provided without separate addressing tests, modification of cell contents, or storage of any test results. The ECS can be stored at the end of the unalterable part of memory. The core memory can also contain the memory test program for check-summing, at the price of a few cells of core storage. The memory test program can also be microprogrammed in the ROM with other test sequences, and this alternative is preferable if sufficient ROM space is available. It has been implicit throughout

---

[12] Different schemes of handling the carry out of the most significant place (e.g., addition to the least significant bit location) reduce the probability of obtaining a proper check sum when failure exists to a negligibly low value.

the foregoing test procedure description that the control and processor units have been tested prior to memory checking so that they can be validly used to calculate the check-sums and do comparisons.

The scratch pad is tested last, and it must be treated somewhat differently, since its contents can change during the mission. Consequently, an ECS could not be calculated and stored earlier for comparison. In addition, there will be some data stored in scratch pad which cannot be destroyed during test mode; e.g., positional data. The same check-sum test technique can, however, still be applied if a small block of scratch pad cells (block A cells in Figure 13) can be altered during test. A like-sized block of stored program cells in the unalterable part of memory (block B cells in Figure 13) is identified and its ECS externally calculated and stored as a constant prior to flight. Figure 13 illustrates the checking procedure for a 1K scratch pad. 256 words of the scratch pad can be altered (block A cells). The sequence of steps to test the 1K scratch pad is listed below:

1. Write contents of block B cells into block A.
2. Check-sum block A and compare to previously stored ECS.
3. Write unalterable scratch pad data of block C cells into block A for temporary storage (block A cells and associated decode modules have been verified by steps 1 and 2).
4. Write contents of block B cells into block C.
5. Check-sum block C and compare to previously stored ECS.
6. Restore data temporarily stored in block A into block C.
7. Continue the procedure with blocks D and E to complete scratch pad test.

Note should be made that the size of block A can be quite small, if necessary, with resulting increase in the number of data shuffles required to complete scratch pad test. Alternate techniques to test

the scratch pad include coding, addition of more hardware, or perhaps acceptance of an untested scratch pad in consonance with reasonable test objectives discussed earlier.

E. TESTING THE CHECKING HARDWARE

The checking hardware represents hard-core circuitry whose proper functioning must be assured before test results are considered valid. The failure of checking circuitry can lead to the very undesirable indication of error when none exists, or failure to flag existing error. To provide assurance that checking hardware is fault-free, one can

1. Provide redundant circuitry with reliability an order of magnitude higher than the circuitry it checks.
2. Provide some earlier periodic check to verify proper operation before test commences.
3. Verify only during periodic maintenance periods.

The first alternative tends to be too expensive, at least doubling the hardware cost of providing built-in test. The third alternative reduces confidence in the test results to an unacceptably low level. A periodic gross functional check of the checking circuitry is probably most feasible, but at the expense of a few words of core storage. Test bit patterns stored in core-memory can be used to initialize the circuitry so that the left and right half-words will differ. Error therefore should be indicated. Identical half-word patterns can be introduced, in which no error should be signalled. Such tests can be made part of the periodic test sequence preceding test of the rest of the computer. While it is recognized that comprehensive test has not been achieved, one can be assured of a high degree of confidence in the checking circuitry for minimal cost and effort.

F. SEQUENCE OF TESTING

The sequence in which testing should be conducted for the parent computer has been indicated in the separate sections. A summary is useful to gain better perspective. For those portions periodically tested, the priority should be:

1. Preflight marginal checks.
2. The checking circuitry (gross functional check).
3. The processor unit.
4. The core memory.
    a. Sense/inhibit function
    b. The core stack (check-sum)
    c. Scratch pad

Those portions tested continuously include:

1. Hard-core housekeeping and service functions (power supplies, clock, and so forth)
2. The control unit
3. Core memory (partially)
    a. Address register
    b. Decode function

G. PROCESSING OF ERROR SIGNALS

Some general comments should be made relative to the handling of error signals once issued. If the goal of providing a separate error signal from each module of the computer is achieved, a large number of sources will be reporting. The reports must be interpreted and processed to achieve the desired test goals.

First, the signal lines should be made "fail-safe"; that is, a voltage should be present on each line except when it is reporting failure. In this way, the line itself is checked since the absence

of a voltage will lead to investigation of the cause. The problem of errors propagating from module to module, giving several false error signals in addition to the accurate error signal, has been resolved locally in the modules by error-inhibit precautions, as in the general processor module checking circuitry. An error signal transmission path should be provided separate from other computer output paths, and by the most direct route to allow signals to be communicated under a failed condition. The problem of signal interpretation remains to be resolved.

A reasonable number of 128 modules with separate error lines will be assumed. By the single failure assumption, only one of the 128 lines will signal error at one time. With 80 pins limiting the 2A NAFI module, two separate error processing modules would be necessary to accommodate the required error inputs. Sixty-four error lines would then input to each module, well within the 80 pin limitation. Encoding circuitry in each module would encode the error source into binary code, each error line having a unique binary number identifying it. Seven output lines, then, would be necessary from each module, six to encode one of 64, and one to indicate which module was sending the encoded error message giving a resolving power of one in 128. A total of 71 input and output lines for each module, plus required power supply and timing inputs, appears reasonable relative to pin limitations. The encoded message would then be routed by direct means to a central buffer register where the message of error location would be preserved by some recording means for later use by maintenance personnel. The message could also be used to turn off the central power

source to avoid the use of contaminated computations. The pilot would be notified of error in accordance with test goals. Care would have to be taken to ensure that failures in checking hardware, detected during pre-test periodic check, did not initiate computer shutdown. In such cases, notification to the pilot that the error checking capability of the computer had failed would allow him to continue its use knowledgeable of the attendant risk.

VI. DESIGN EVALUATION

Any test design is subject to the unique limitations imposed by the parent design program, and the example used was no exception. The design presented achieves the reasonable objectives established for it in almost all instances:

1. A thorough self-test capability is provided for the parent computer in the airborne environment with a high confidence level for the test results. The risk of undetected error is kept negligibly low.
2. The test design represents a unique series of tradeoffs, optimizing the test performance per dollar for the short schedule, low risk program. Maximum advantage was taken of proposed architectural characteristics for the machine. The hardware-software split duplication technique and the proposed modification of an existing memory design illustrate this.
3. Partitioning of the computer was achieved using the specified NAFI 2A module. Detection and isolation of the most important classes of faults to this modular level is automatically provided. This capability was achieved while allowing for flexible word length with minimal basic design changes. In the highly regular processor and memory units, the number of different module types was kept favorably low.
4. Redundancy was not generally used. The capability of significant test performance is provided for considerably less than duplication of hardware.
5. The test design required very few cells of core storage, such requirements being limited to a few constants and possibly a memory test routine of short length. A simple pseudo-random number generator to provide test but patterns was substituted for a large number of stored constants. The coding techniques used required no core storage, leaving maximum word length available for operational use. Dissociating the core memory from the processor and control units simplified the overall test problem.
6. Operational degradation was minimized throughout. An interruptable microprogrammed routine using idle time and executed at read only memory cycle speed provides valid test without infringing on operational availability.

Assignment of a specific figure of merit to the test design must await choice of specific hardware, and the important microprogramming of the test routine upon which much of the potential test performance is predicated.

Various figures of merit can be assigned to a test design. Davis [Ref. 9] developed a formula to assign a figure of merit to his residue code arithmetic unit test scheme. Other figures relating to cost, such as the 10% added hardware figure mentioned earlier, or in more absolute terms the cost of BIT per gate tested have been assigned. The ultimate justification for a self-test capability is its measured performance in detecting errors. A high confidence level that a high percentage of potential failure sources have been checked seems to the author to be the best figure of merit.

Evaluation of a self-test capability can be accomplished in several ways. One technique which allows such evaluation is simulation, during which faults can be artificially duplicated to verify expected test response. Once the computer is built, actual faults can be injected and the response measured. Failure history for a production machine can also help in evaluating test efficiency. A full-scale simulation of the parent computer with self-test circuitry was envisioned.

The example design promises to provide significantly more test capability per dollar than previous designs for similar computers. Its potential beneficial effect on overall cost of ownership makes the self-test capability provided by the design a very attractive feature. Recognition of this fact should certainly result in greater future emphasis on the relatively new field of built-in self-test design.

VII. SUGGESTED FURTHER INVESTIGATION

The subject of derivation of an optimal test routine using the micro-instruction set is an interesting one for future work. Many techniques, some briefly presented herein, suggest ways in which the states of a block of logic can be identified and related to the micro-instructions. Additionally, special instructions for test use only can be formulated, as needed. Computer-aided design fits well in this category.

Once error signals from each module can be provided, the subject of automatic reconfiguration for continued operation after failure can be addressed. Ideally, the error signal from a "bad" module would be used to turn off the bad module and switch in a substituting module. For example, in the processor, the three identical modules of a set could be joined by a fourth identical module to be used in the event of failure. The ability to add such a reconfiguration capability in modular form might prove to be an attractive option available at extra cost dependent on the computer's intended use.

The ability of a computer to continue to operate after failure in a degraded mode using its remaining unfailed circuitry might be investigated. For example, limited operations might continue at a slower speed for high priority tasks related to aircraft survival (e.g., electronic countermeasures and navigation.)

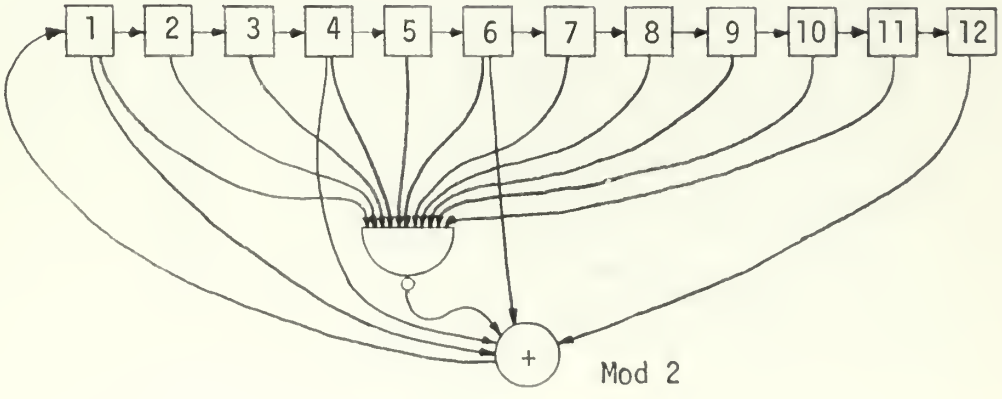
Lastly, the effects of continued technological advance on test design and self-repair offer fruitful subjects for further investigation.

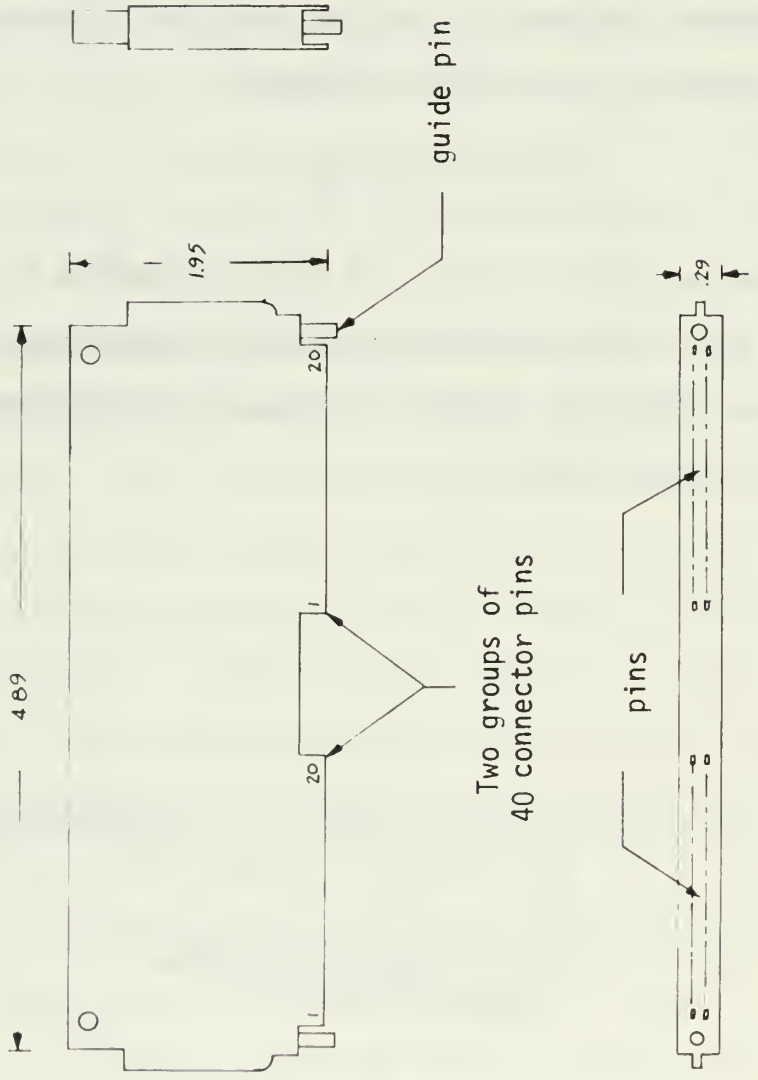
APPENDIX A — PSEUDO-RANDOM NUMBER GENERATOR

The pseudo-random number generator shown below generates the maximum length sequence of 2^{12} different 12-bit binary patterns. The numbers so produced are random in each bit position. The generator implements the modulo 2 irreducible polynomial

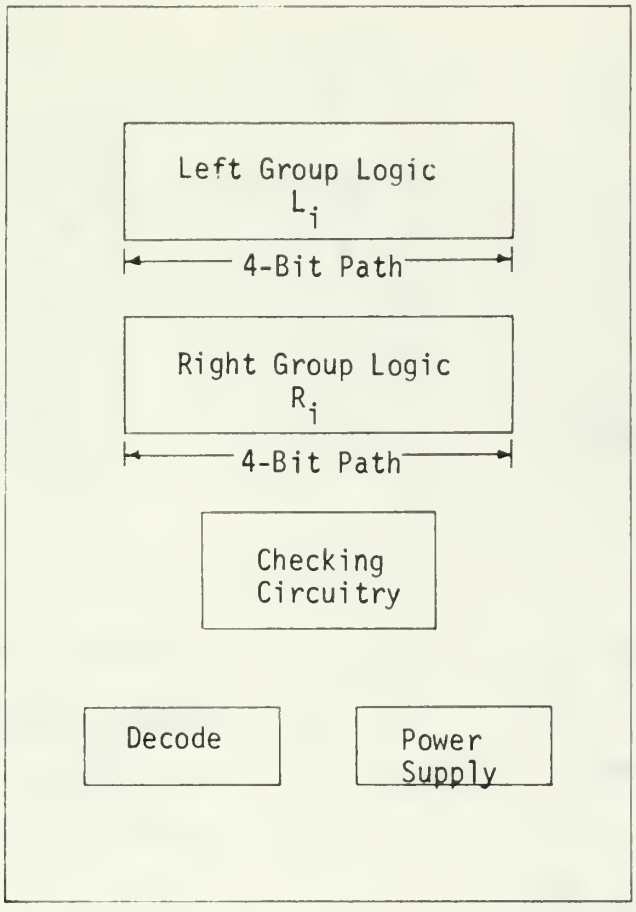
$$x^{12} + x^6 + x^4 + x + 1$$

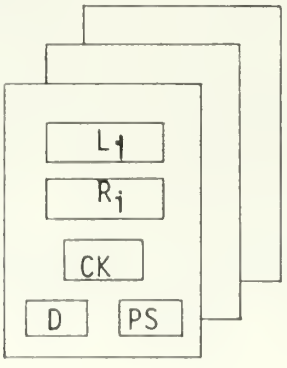
as a linear feedback shift register. A different pattern is produced at each clock pulse. The nonlinearity of the all-zero case is added by the 11 - input NAND gate (which, of course, can be implemented as several gates instead of one).



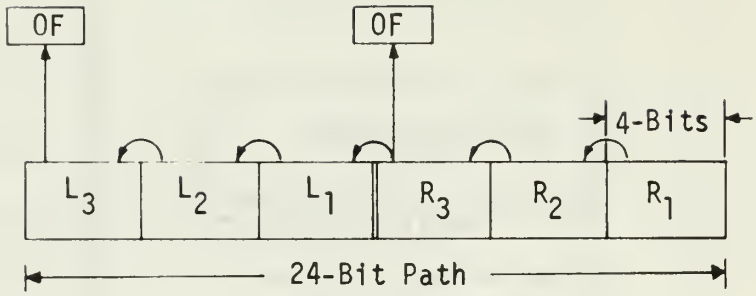


The 2A NAFI Module
Figure 1



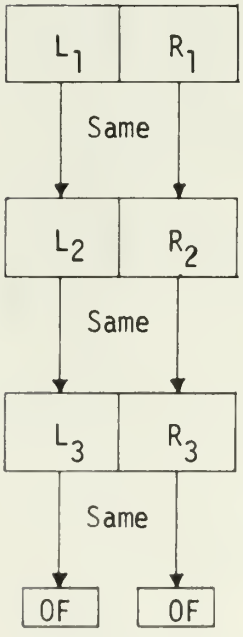


General Processor Module
Figure 2



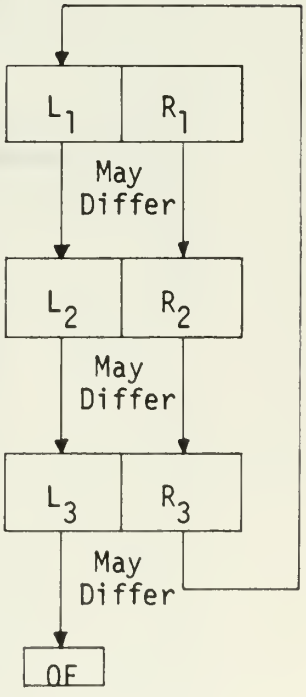
(a)

OF=Overflow



Carries in
Half-Word Operations

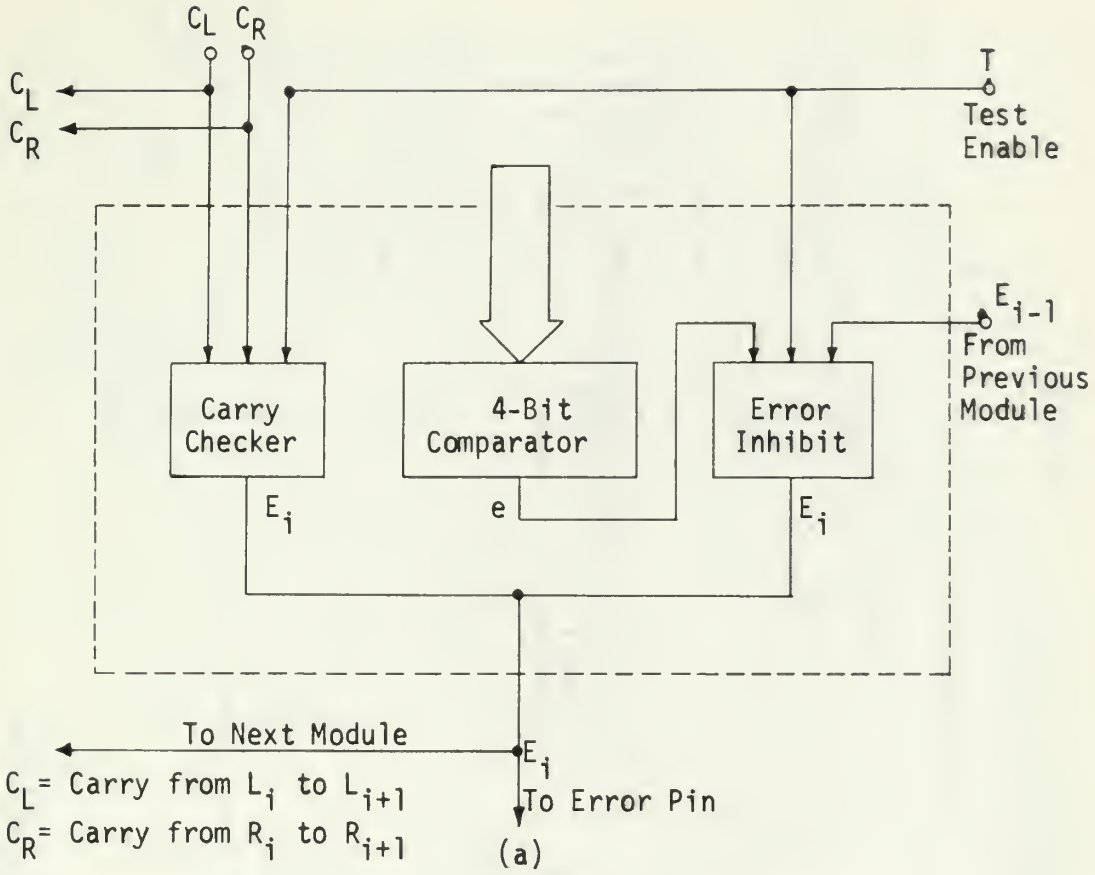
(b)



Carries in
Full-Word Operations

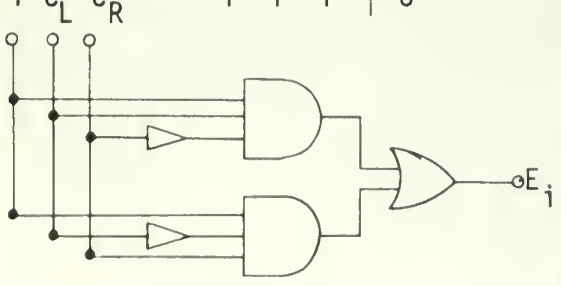
(c)

Carries
Figure 3

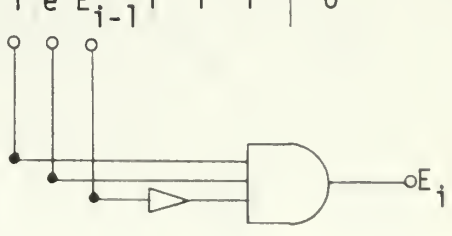


T	C_L	C_R	E_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	0

T	e	E_{i-1}	E_i
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	1
1	1	1	0

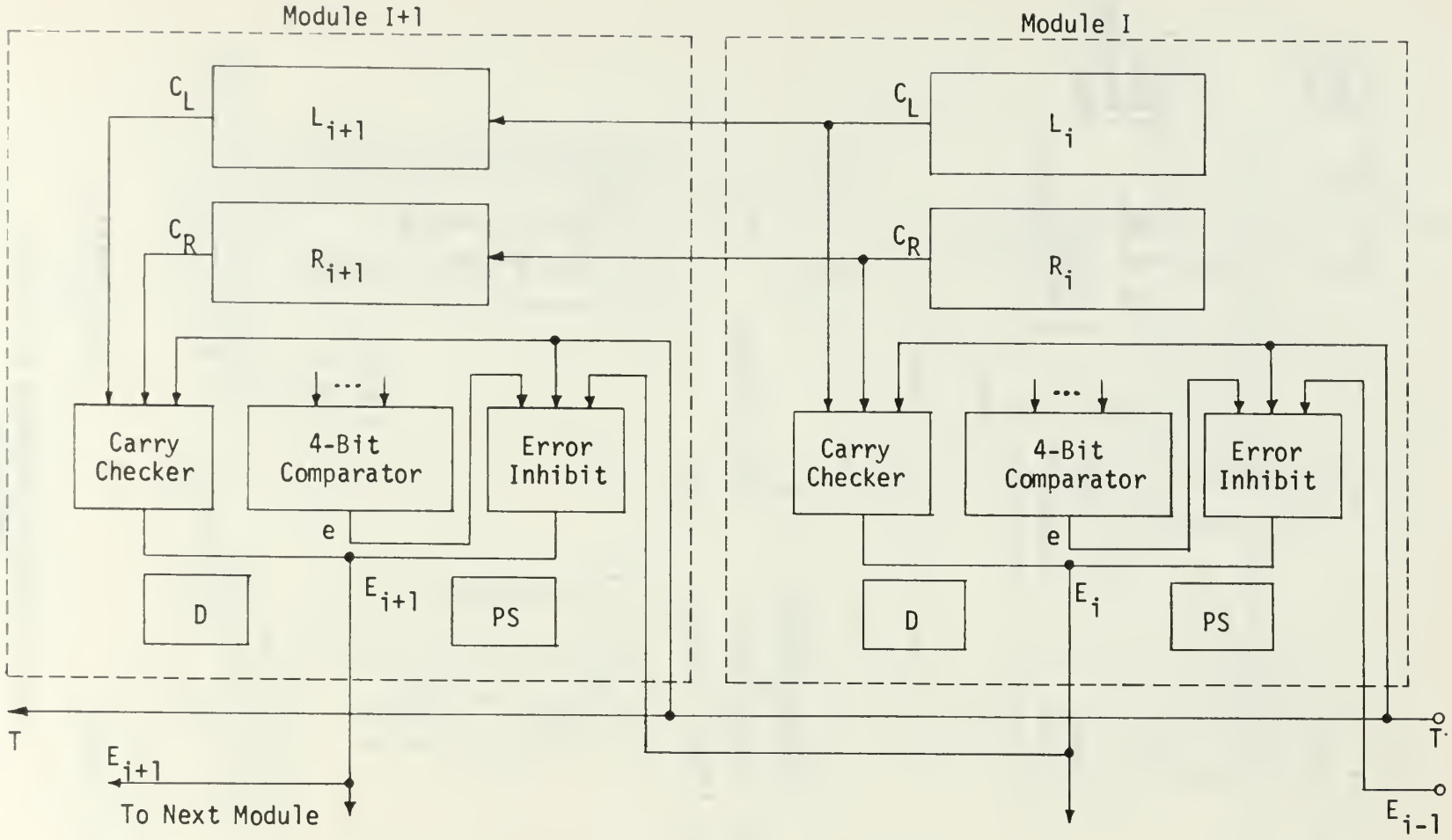


Carry Checker
(b)

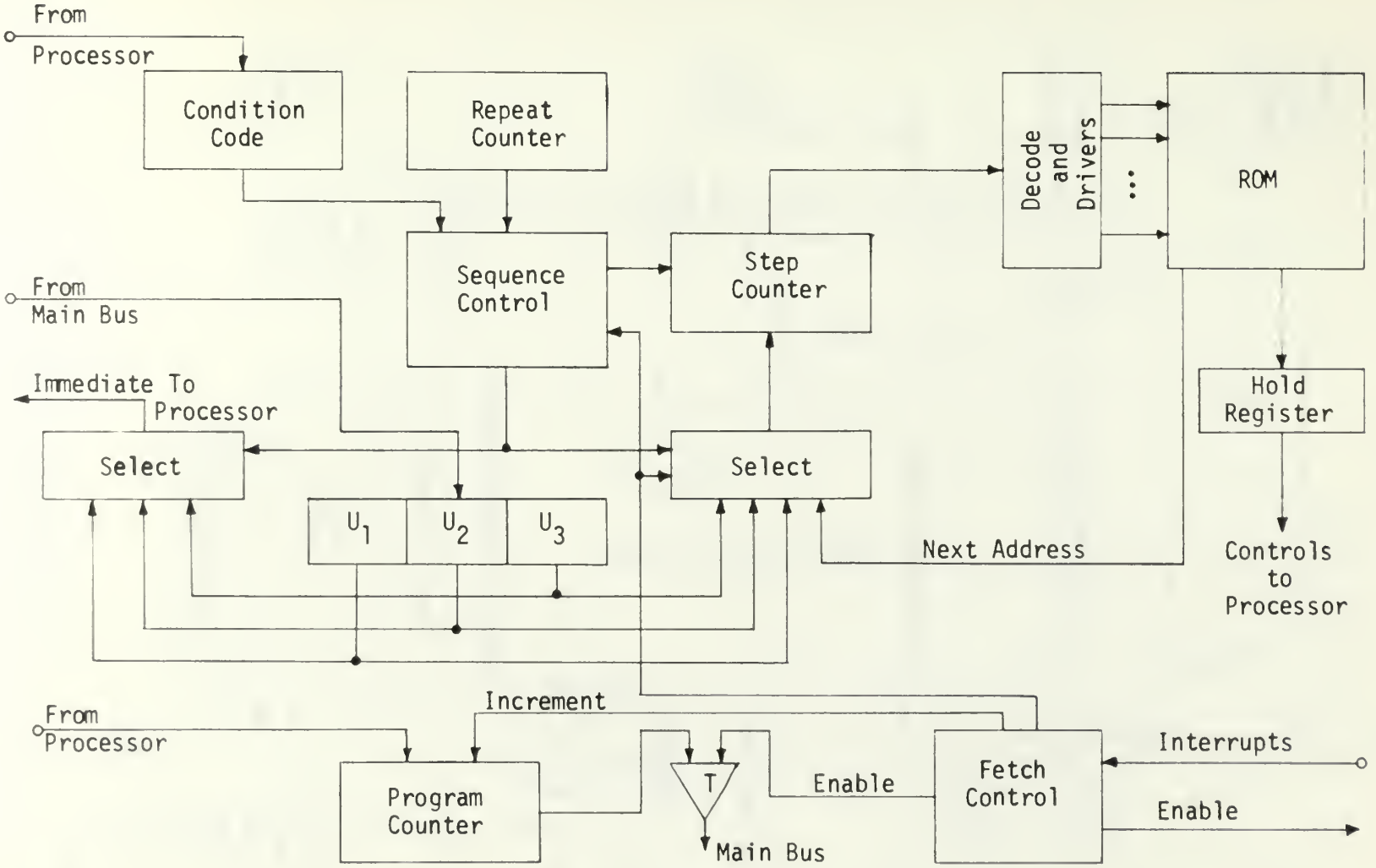


Error Inhibit
(c)

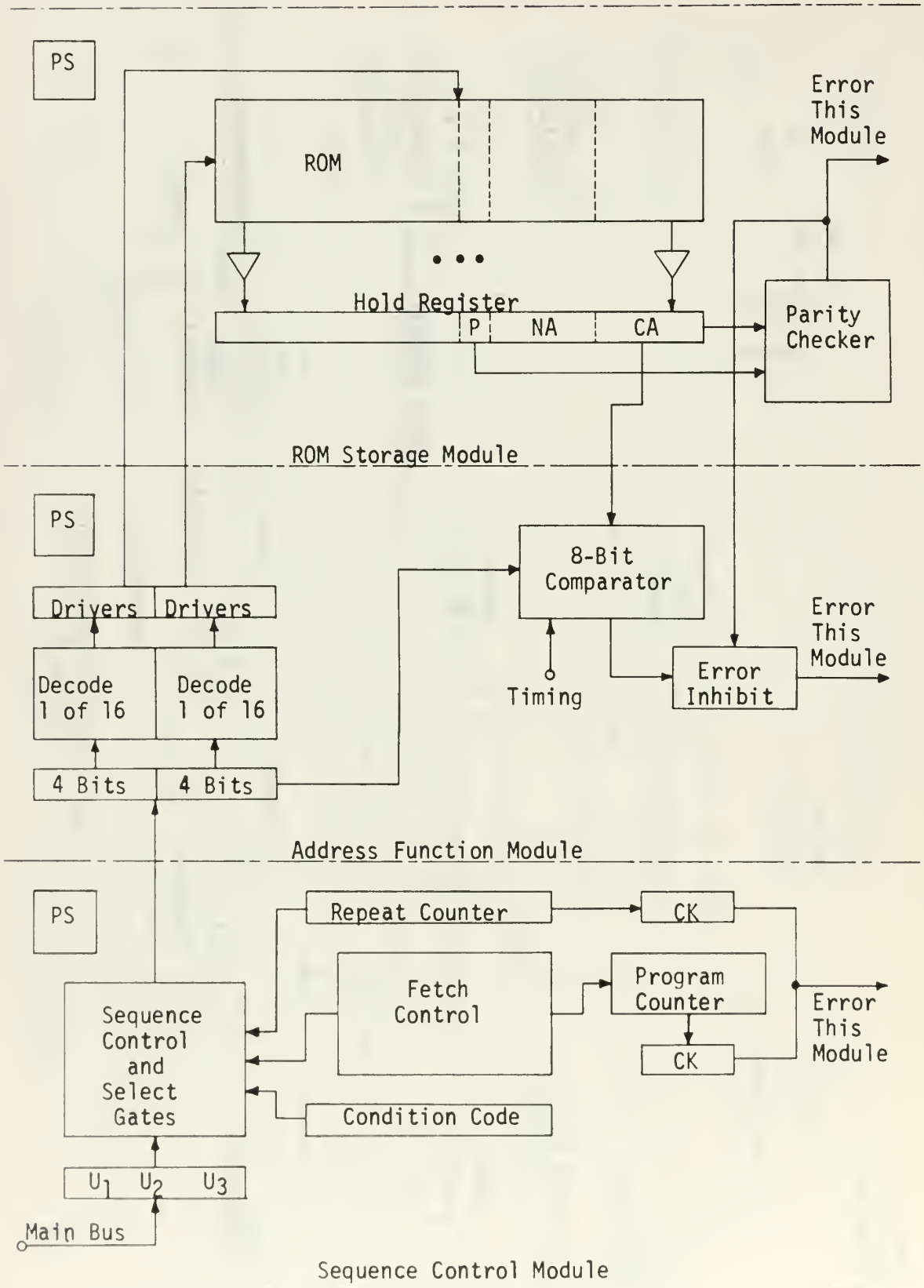
Processor Module Checking Circuitry
Figure 4



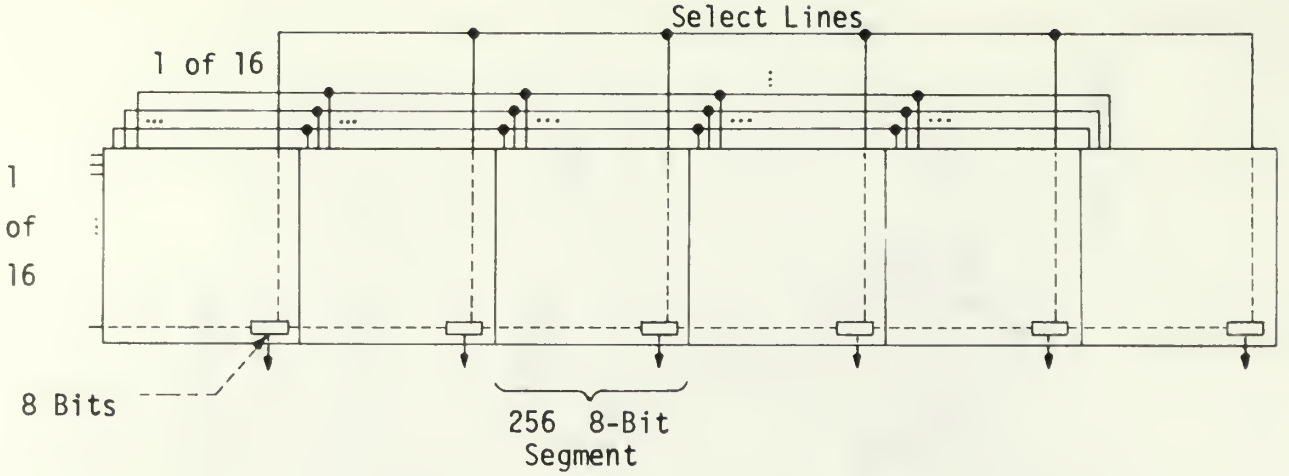
Modular Relationships
Figure 5



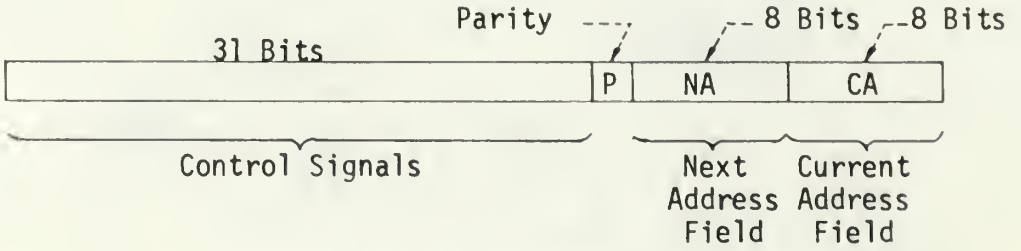
Control Unit Organization
Figure 6



The Partitioned Control Unit
Figure 7

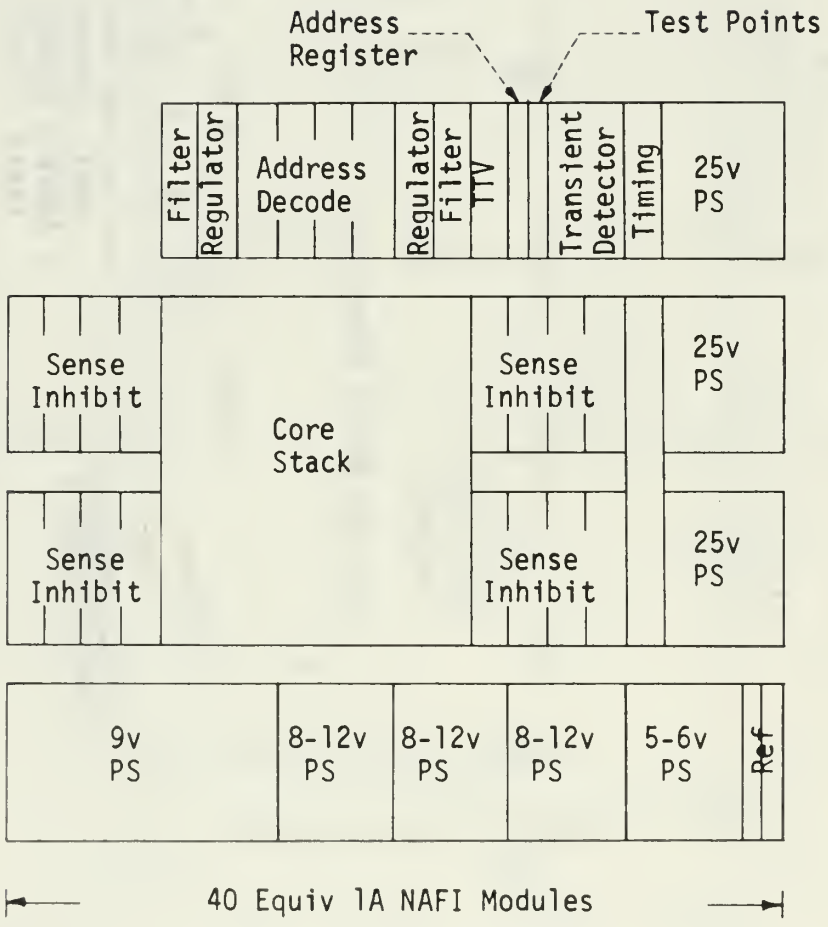


ROM 256 48-Bit Words

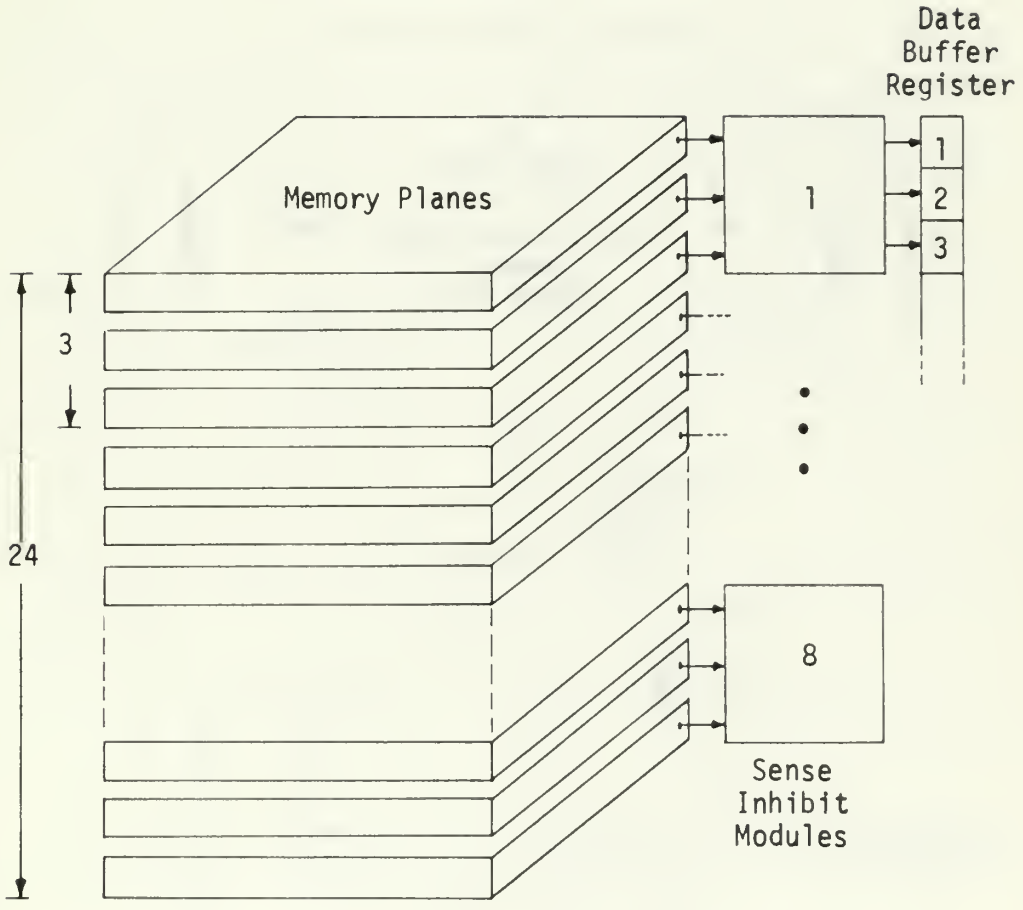


Test Fields of the Microword

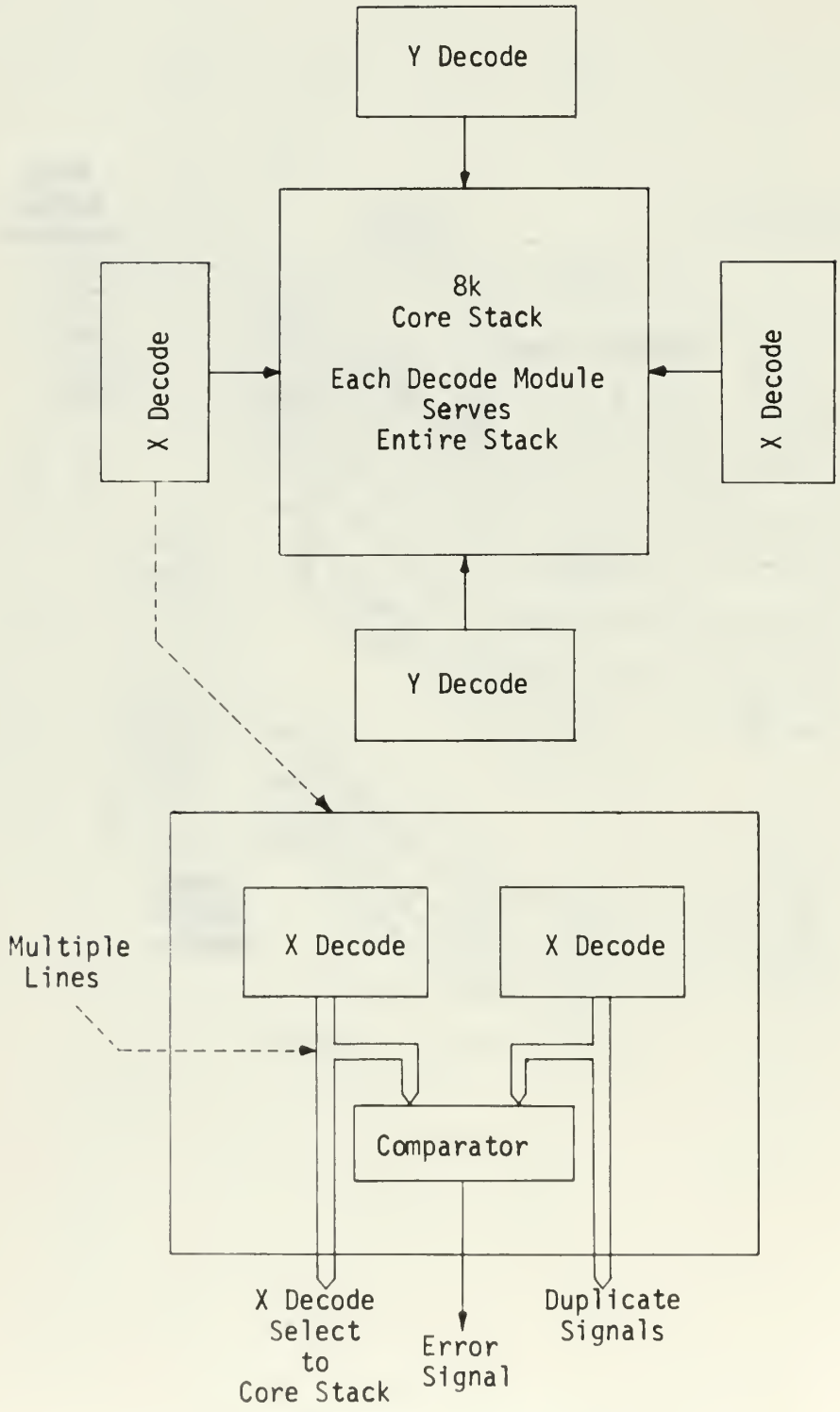
ROM Storage Module Details
Figure 8



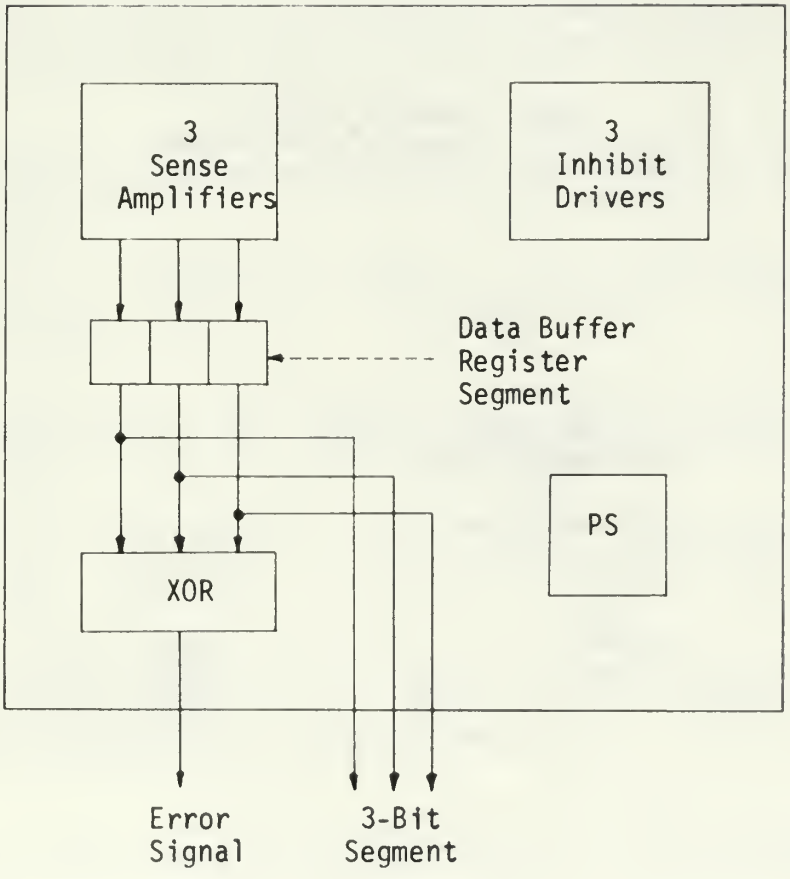
Standard Memory Unit Layout
Figure 9



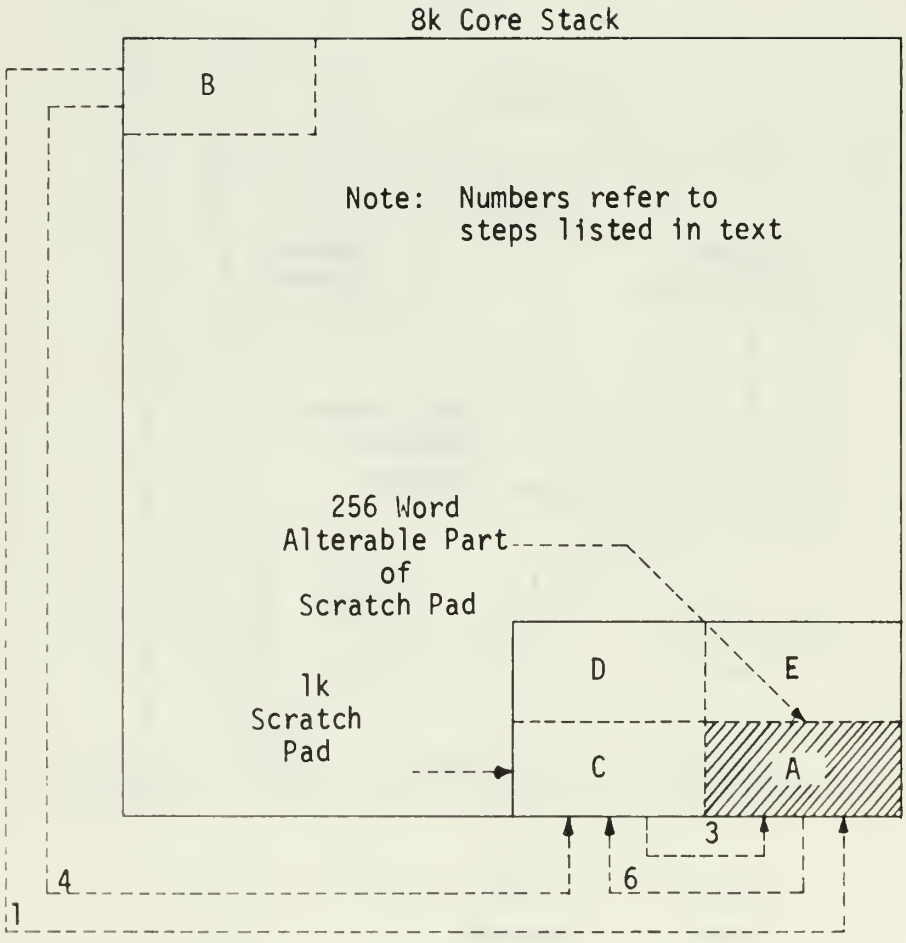
Sense/Inhibit Related to Core Stack
Figure 10



Decode Module
Figure 11



Sense/Inhibit Module
Figure 12



Scratch Pad Test Procedure
Figure 13

BIBLIOGRAPHY

1. Armstrong, D. B., "On Finding a Nearly Minimal Set of Fault Detection Tests for Combinational Logic Nets," IEEE Transactions on Electronic Computers, v. EC-15, p. 66-73, February 1966.

2. Ball, Michael and Hardie, Fred, "Self-Repair in a TMR Computer," Computer Design, v. 8, p. 54-57, February 1969.

3. Bashkow, T. R., Friets, J., and Karson, A., "A Programming System for Detection and Diagnosis of Machine Malfunctions," IEEE Transactions on Electronic Computers, v. EC-12, p. 10-17, February 1963.

4. Brauer, J. B., "The Numbers Game — Who Wins?" paper presented at Annual Symposium on Reliability, Washington, D. C., 10 January 1967.

5. Carter, W. C., and others, "Design of Serviceability Features for the IBM System 360," IBM Journal, v. 8, p. 115-126, April 1964.

6. Chang, Herbert Y., "An Algorithm For Selecting an Optimum Set of Diagnostic Tests," IEEE Transactions on Electronic Computers, v. EC-14, p. 706-711, October 1965.

7. Cohen, J. J. and Whitaker, L. A., "Improved Techniques in Diagnostic Programming," The Sylvania Technologist, v. 13, p. 90-96, July 1960.

8. Connolly, J. B. and Schmidt, W. G., "Failure Erasure Circuitry; A Duplicative Technique of Failure - Masking Systems," IEEE Transactions on Electronic Computers, v. EC-16, p. 82-85, February 1967.

9. Davis, R. A., "A Checking Arithmetic Unit," AFIPS Proceedings Fall Joint Computer Conference, 1965, v. 27, part 1, p. 705-713.

10. Department of the Navy NAVWEPS OD 30355 (3D Revision), Standard Hardware Program Data Handbook, v. 1, table 4-2, figure 4-3, 1 August 1968.

11. Downing, R. W., Nowak, J. S., and Tuomenoksa, L. S., "No. 1 ESS Maintenance Plan," Bell System Technical Journal, v. 43, p. 1961 - 2019, September 1964.

12. Eldred, R. D., "Test Routines Based on Symbolic Logic Statements," Journal of ACM, v. 6, p. 33-36, January 1959.

13. Forbes, R. E., Rutherford, D. H., and Stieglitz, C. B., "A Self-Diagnosable Computer," AFIPS Proceedings Fall Joint Computer Conference, 1965, v. 27, part 1, p. 1073-1086.

14. Galey, J. M., Norby, R. E., and Roth, J. P., "Techniques for the Diagnosis of Switching Circuit Failures," IEEE Transactions on Communications and Electronics, v. 83, p. 509-514, September 1964.

15. Golomb, S. W., Shift Register Sequences, Holden-Day, 1967.

16. Hackl, F. J. and Shirk, R. W., "An Integrated Approach to Automated Computer Maintenance," Proceedings Annual IEEE Symposium on Switching Circuits and Logical Design, 6th, Ann Arbor, Michigan, 1965, p. 280-302.

17. Hamming, R. W., "Error Detecting and Error Connecting Codes," The Bell System Technical Journal, v. 26, p. 147-160, April 1950.

18. Happ, W. A., "Combinatorial Analysis of Multi-Terminal Devices," IEEE Transactions on Systems Science and Cybernetics, v. SSC-3, p. 21-27, June 1967.

19. Hausrath, D. A. and Fleming, D. C., "Reducing Failure Rate Prediction Uncertainties," Proceedings Annual Symposium on Reliability, Boston, Massachusetts, 1968, v. 1, p. 226-235.

20. Hennie, F. C., "Fault Detecting Experiments for Sequential Circuits," Proceedings, Annual IEEE Symposium on Switching Circuits and Logical Design, 5th, Princeton, N. J., October 1964, p. 95-110.

21. Joseph, E. C., "Impact of Large Scale Integration on Aerospace Computers," IEEE Transactions on Electronic Computers, v. EC-16, p. 558-561, October 1967.

22. Kautz, William H., "Fault Testing and Diagnosis in Combinational Digital Circuits," IEEE Transactions on Computers, v. C-17, p. 352-366, April 1968.

23. Kirkman, R. A., "Failure Concepts in Reliability Theory," IEEE Transactions on Reliability, v. R-12, p. 1-10, December 1963.

24. Kirkman, R. A., "Failure Prediction in Electronic Systems," IEEE Transactions on Aerospace and Electronic Systems, v. AES-2, p. 700-707, November 1966.

25. Kohavi, Zvi and Lavallee, Pierre, "Design of Diagnosable Sequential Machines," AFIPS Proceedings Spring Joint Computer Conference, 1967, v. 30, p. 713-718.

26. LaMacchia, S. E., "Diagnosis in Automatic Checkout," IRE Transactions on Military Electronics, v. 6, p. 302-309, July 1962.

27. Lee, Fred, "An Automatic Self-Checking and Fault-Locating Method," IRE Transactions on Electronic Computers, v. EC-11, p. 649-654, October 1962.

28. Lowrie, R. W., "High Reliability Computers using Duplex Redundancy," Electronic Industries, p. 116-128, August 1963.

29. Lynn, D. K., Meyer, C. S., and Hamilton, D. J., eds., Analysis and Design of Integrated Circuits, p. 6, McGraw-Hill, 1967.

30. Maling, K. and Allen, E. L., "A Computer Organization and Programming System for Automated Maintenance," IEEE Transactions on Electronic Computers, v. EC-12, p. 887-895, December 1963.

31. Manning, Eric, "On Computer Self-Diagnosis Part I — Experimental Study of a Processor," IEEE Transactions on Electronic Computers, v. EC-15, p. 873-881, December 1966.

32. Manning, Eric, "On Computer Self-Diagnosis Part II — Generalizations and Design Principles," IEEE Transactions on Electronic Computers, v. EC-15, p. 882-890, December 1966.

33. Merwin, R. E., "Processor Control Checkup System," IBM Technical Disclosure Bulletin, v. 8, p. 58-59, June 1965.

34. Minner, E. S. and Romero, H. A., "Reliability Testing of F-111A Avionics System," Proceedings Annual Symposium on Reliability, Boston, Massachusetts, 1968, v. 1, p. 567-575.

35. National Aeronautics and Space Administration Report CR-65254, AES-EPO Study Program Final Report, by IBM Corp., Federal Systems Division, Oswego, N. Y., v. 2, p. 135-166, 31 December 1965.

36. Peterson, W. W., Error-Correcting Codes, MIT Press and Wiley, 1961.

37. Poage, J. F. and McClusky, E. J. Jr., "Derivation of Optimum Test Sequences for Sequential Machines," Proceedings, Annual IEEE Symposium on Switching Circuits and Logical Design, 5th, Princeton, N. J., October 1964, p. 121-132.

38. Preparata, F. P., Metze, Gernot, and Chien, R. T., "On the Connection Assignment Problem of Diagnosable Systems," IEEE Transactions on Electronic Computers, v. EC-16, p. 848-854, December 1967.

39. Rao, T. R. N., "Error-Checking Logic for Arithmetic-Type Operations of a Processor," IEEE Transactions on Computers, v. C-17, p. 845-849, September 1968.

40. Reference Data for Radio Engineers, 5th ed., p. 40-20, Howard W. Sams, 1968.

41. Roth, J.P., "Diagnosis of Automata Failures: A Calculus and A Method," IBM Journal, v. 10, p. 278-291, July 1966.

42. Sellers, F.F., Jr., Hsiao, M.Y., and Bearnson, L.W., "Analyzing Errors with Boolean Difference," IEEE Transactions on Computers, v. C-17, p. 676-683, July 1968.

43. Sellers, F.F., Jr., Hsiao, M.Y., and Bearnson, L.W., Error Detecting Logic for Digital Computers, McGraw-Hill, 1968.

44. Seshu, Sundaram, "On An Improved Diagnosis Program," IEEE Transactions on Electronic Computers, v. EC-14, p. 76-79, February 1965.

45. Seshu, S., and Freeman, D.N., "The Diagnosis of Asynchronous Sequential Switching Systems," IRE Transactions on Electronic Computers, v. EC-11, p. 459-465, August 1962.

46. Terris, I. and Melkanoff, M.A., "Investigation and Simulation of A Self-Repairing Digital Computer," Proceedings Annual IEEE Symposium on Switching Circuits and Logical Design, 6th, Ann Arbor, Michigan, 1965, p. 264-278.

47. Tsiang, S.H. and Ulrich, W., "Automatic Trouble Diagnosis of Complex Logic Circuits," Bell System Technical Journal, v. 41, p. 1177-1200, July 1962.

48. Weber, Samuel, "LSI: the Technologies Converge," Electronics, v. 38, p. 124-129, 20 February 1967.

49. Weitzenfeld, A.S. and Happ, W.W., "Combinatorial Techniques for Fault Identification in Multiterminal Networks," IEEE Transactions on Reliability, v. R-16, p. 93-99, December 1967.

INITIAL DISTRIBUTION LIST

		No. Copies
1.	Defense Documentation Center Cameron Station Alexandria, Virginia 22314	20
2.	Library, Code 0212 Naval Postgraduate School Monterey, California 93940	2
3.	Associate Professor M. L. Cotton, Code 52Cc Department of Electrical Engineering Naval Postgraduate School Monterey, California 93940	1
4.	Major Edward O. Bierman, USMC 14915 Daytona Court Woodbridge, Virginia 22191	1
5.	Mr. Charles A. Pullen Mail Station 6/E-102 Hughes Aircraft Company Culver City, California 90230	1
6.	Commandant of the Marine Corps (Code A03C) Headquarters, U.S. Marine Corps Washington, D.C. 20380	1
7.	James Carson Breckinridge Library Marine Corps Development and Educational Command Quantico, Virginia 22134	1

Security Classification

DOCUMENT CONTROL DATA - R & D

Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

ORIGINATING ACTIVITY (Corporate author)	2a. REPORT SECURITY CLASSIFICATION
Naval Postgraduate School Monterey, California 93940	Unclassified
	2b. GROUP

REPORT TITLE

Built-In Self-Test for an Airborne Digital Computer

DESCRIPTIVE NOTES *(Type of report and inclusive dates)*

Master's Thesis; June 1969

AUTHOR(S) *(First name, middle initial, last name)*

Edward Oliver Bierman

REPORT DATE	7a. TOTAL NO. OF PAGES	7b. NO. OF REFS
June 1969	97	49

8a. CONTRACT OR GRANT NO	9a. ORIGINATOR'S REPORT NUMBER(S)
b. PROJECT NO	
c.	9b. OTHER REPORT NO(S) *(Any other numbers that may be assigned this report)*
d.	

10 DISTRIBUTION STATEMENT

Distribution of this document is unlimited.

11 SUPPLEMENTARY NOTES	12. SPONSORING MILITARY ACTIVITY
	Naval Postgraduate School Monterey, California 93940

13 ABSTRACT

This thesis describes the design of a built-in self-test capability for a military airborne digital computer. The supportive investigation of program constraints and their effects on the example test design is intended to give broad perspective to the general self-test design problem. Alternate procedures for achieving the goal of airborne detection and isolation of a certain class of failures to the modular level are surveyed. A specific test design is evolved illustrating the unique mix of program-oriented, periodic techniques, and added hardware, continuous techniques best suited to the example development program. The test design is evaluated and further work is suggested.

DD FORM 1 NOV 68 1473 (PAGE 1)

S/N 0101-807-6811

Security Classification

A-31408

14 KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Computer Self-Test						
Failure Diagnosis						
Automatic Test						
Modular Computer						

DD FORM 1 NOV 65 1473 (BACK)

S/N 0101-807-6801

A-3140